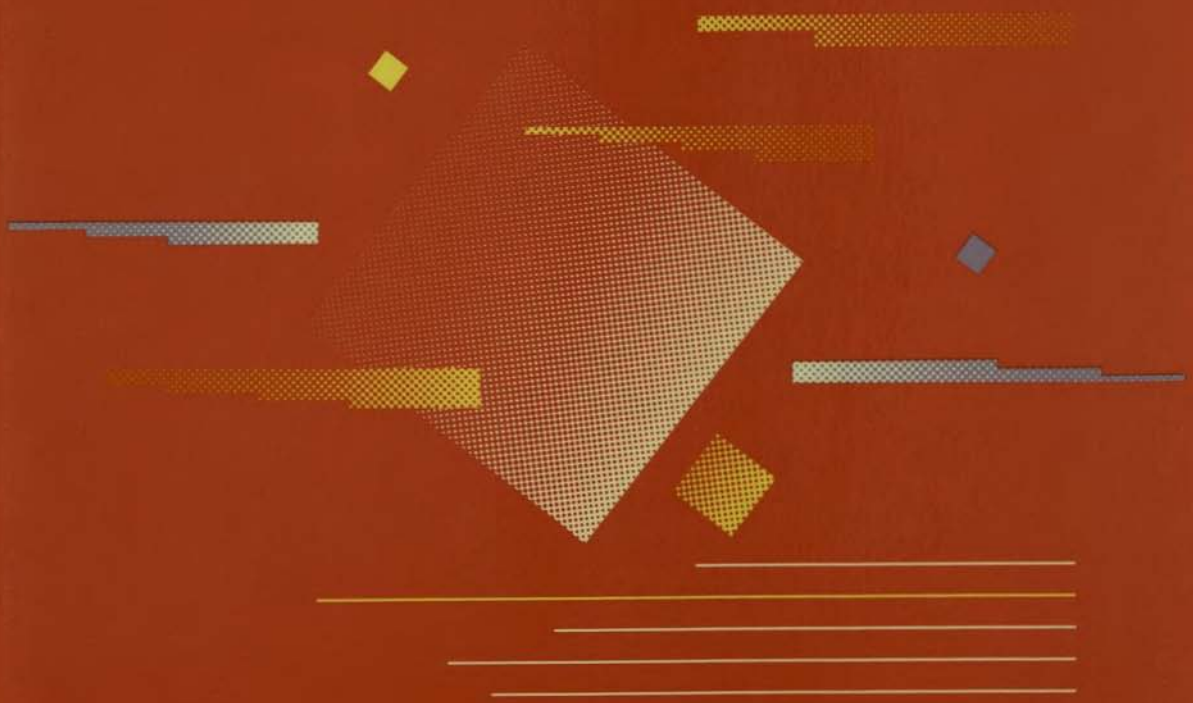
INTERGRAPH
Advanced Processor Division

CLIPPER™
32-Bit Microprocessor

User's Manual



Overview	1
CLIPPER Programming Model	2
Memory Management	3
Instruction and Data Caches	4
Pipelining	5
Floating-Point Unit	6
CLIPPER Bus	7
Hardware Design Considerations	8
Software Design	9
Instruction Set	10
CLIPPER Module Signal Descriptions	A
Instruction Set Summary	B
Questions and Answers	C
Index	I

FAIRCHILD

A Schlumberger Company

CLIPPER™

32-Bit Microprocessor

User's Manual

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

The reader is granted permission to copy the computer programs contained herein. The programs have been tested on CLIPPER™ hardware and operate correctly. Fairchild makes no warranty of any kind with respect to their use, including no warranty against errors or against infringement. The reader assumes all responsibility for use.

The following are trademarks of Fairchild Semiconductor Corporation:
CLIPPER
CLIX

UNIX is a trademark of AT&T Bell Laboratories.
VAX, ULTRIX and Q-bus are trademarks of Digital Equipment Corporation.
MULTIBUS is a trademark of Intel Corporation.
VME Bus is a trademark of Motorola Inc.

© 1987 Fairchild.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-138058-3 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Table of Contents

Preface

Chapter 1. OVERVIEW

1.1 Introduction 1-1
1.2 Why a New Architecture? 1-1
1.2.1 The Architectural Performance Gap 1-2
1.2.2 Compatibility—Preserving Your Software Investment 1-3
1.3 High Performance 1-4
1.3.1 Advanced Cache Management 1-5
1.3.2 Increasing Bus Bandwidth 1-6
1.3.3 Sophisticated Pipelining 1-6
1.3.4 Integrated Processing Units 1-6
1.4 Streamlined Instruction Set 1-7
1.5 Virtual Memory and Operating System Mechanisms 1-8
1.6 Software and Support 1-9
1.7 System Architecture 1-9

Chapter 2. CLIPPER PROGRAMMING MODEL

2.1 Introduction 2-1
2.2 Register Sets 2-1
2.2.1 User And Supervisor Registers 2-2
2.2.2 Floating-Point Registers 2-2
2.2.3 Program Status Word (PSW) 2-2
2.2.4 System Status Word (SSW) 2-5
2.2.5 CAMMU Registers 2-6
2.3 Data Types 2-7
2.4 Instructions 2-7
2.4.1 Instruction Formats—No Address 2-10
2.4.2 Instruction Formats—With Address 2-10
2.4.3 Addressing Mode Applications 2-15
2.5 Supervisor and User Modes of Operation 2-16
2.6 Exceptions 2-16
2.6.1 Traps 2-19
2.6.1.1 Data Memory Trap Group 2-19
2.6.1.2 Floating-Point Arithmetic Trap Group 2-20
2.6.1.3 Integer Arithmetic Trap Group 2-21
2.6.1.4 Instruction Memory Trap Group 2-21
2.6.1.5 Illegal Operation Trap Group 2-21
2.6.1.6 Diagnostic Trap Group 2-22
2.6.1.7 Supervisor Call Traps 2-22
2.6.1.8 Multiple Traps 2-22
2.6.2 Interrupts 2-22
2.6.2.1 Maskable Interrupt Request/ Acknowledge Protocol 2-22
2.6.2.2 Non-Maskable Interrupts 2-23

Chapter 3. MEMORY MANAGEMENT

3.1 Introduction 3-1
3.2 Memory Architecture 3-1
3.2.1 Real Address Space 3-1
3.2.2 Virtual Address Space 3-2
3.3 Address Translation 3-2
3.3.1 CLIPPER Two-Level, Page-Based Mapping 3-4
3.4 Virtual Memory 3-10
3.5 CLIPPER MMU Implementation 3-11
3.5.1 HTLB 3-12
3.5.2 CAMMU Register Descriptions 3-16
3.5.2.1 Supervisor PDO Register 3-16
3.5.2.2 User PDO Register 3-16
3.5.2.3 Fault Register 3-16
3.5.2.4 Control Register 3-16
3.5.3 CAMMU Register Access 3-17
3.5.3.1 CAMMU Register Addressing 3-18
3.5.3.2 CAMMU Data Format 3-19

Chapter 4. INSTRUCTION AND DATA CACHES

4.1 Introduction 4-1
4.2 Cache Access Time 4-2
4.3 Hit Rate 4-2
4.4 Replacement Time 4-5
4.5 CLIPPER Cache Implementation 4-7

Chapter 5. PIPELINING

5.1 Introduction 5-1
5.2 Three-Phase Pipeline 5-2
5.3 Dual Execution Units 5-3

Chapter 6. FLOATING-POINT UNIT

6.1 Introduction 6-1
6.2 Programming Model 6-1
6.2.1 Registers 6-1
6.2.1.1 Floating-Point Registers 6-1
6.2.1.2 PSW 6-2

6.2.1.3 SSW 6-3
6.2.1.4 Floating Exception Registers 6-3
6.2.2 Instructions and Data Types 6-3
6.2.3 Initialization 6-3
6.3 Data Movement Operations 6-4
6.4 Integer Operations 6-5
6.4.1 Integer Multiply 6-5
6.4.2 Integer Divide and Mod 6-6
6.5 Overview of the IEEE Standard 754 For Binary Floating-Point Arithmetic 6-6
6.5.1 Formats 6-7
6.5.2 Encoding of Values 6-8
6.5.3 Rounding 6-9
6.5.4 Operations 6-10
6.5.5 Infinity and NaN Arithmetic 6-10
6.5.6 Traps and Exceptions 6-10
6.6 Floating-Point Arithmetic Operations 6-10
6.6.1 Floating-Point Formats 6-10
6.6.1.1 Single-Precision Format 6-12
6.6.1.2 Double-Precision Format 6-15
6.6.1.3 Not A Number 6-18
6.6.1.4 Infinities 6-19
6.6.2 Addition and Subtraction 6-19
6.6.3 Multiplication 6-23
6.6.4 Division 6-25
6.6.5 Comparison 6-27
6.6.6 Floating-Point Remainder, Square Root and Round to Integer 6-27
6.6.6.1 Floating-Point Round to Integer in Floating-Point Format 6-28
6.6.6.2 Floating-Point Square Root 6-28
6.6.6.3 Floating-Point Remainder 6-28
6.7 Floating-Point Format Conversion Instructions 6-29
6.7.1 Conversions Between Floating-Point Formats 6-29
6.7.1.1 Convert Double to Single 6-29
6.7.1.2 Convert Single to Double 6-30
6.7.2 Conversion Between Floating-Point and Integer Formats 6-30
6.7.2.1 Convert Double Floating to Word: IEEE Standard Version 6-30
6.7.2.2 Convert Truncating Double Floating to Word 6-31
6.7.2.3 Convert Rounding Double Floating to Word 6-31
6.7.2.4 Convert Single Floating to Word: IEEE Standard Version 6-31
6.7.2.5 Convert Truncating Single Floating to Word 6-32
6.7.2.6 Convert Rounding Single Floating to Word 6-32
6.8 IEEE 754 Appendix Operations 6-33
6.8.1 Floating-Point Scale By an Integer Power of Two 6-33
6.8.2 Negate Floating Register 6-33
6.8.3 Copy Sign 6-33
6.8.4 Get Binary Exponent 6-33
6.8.5 Successor Function 6-33
6.8.6 Classification Functions 6-34
6.9 FPU Exceptions 6-34
6.9.1 Integer Exceptions 6-34
6.9.1.1 Integer Divide by Zero 6-34
6.9.1.2 Integer Overflow 6-34
6.9.2 Floating-Point Exceptions 6-34
6.9.2.1 Floating Overflow 6-35
6.9.2.2 Floating Underflow 6-36
6.9.2.3 Floating Divide By Zero 6-36
6.9.2.4 Floating Invalid Operation 6-36
6.9.2.5 Floating Inexact 6-36

Chapter 7. CLIPPER BUS

7.1 Introduction 7-1
7.2 Signal Descriptions 7-1
7.2.1 Address/Data Bus 7-4
7.2.2 Clocks 7-4
7.2.3 Cycle Type 7-4
7.2.4 Memory Space System Tags 7-5
7.2.5 Bus Arbitration Signals 7-6
7.2.6 Bus Control Signals 7-6
7.2.7 Interrupt Bus And Control 7-7
7.2.8 Error Signals 7-8
7.2.9 Reset and Diagnostics Control 7-9
7.2.9.1 Self-Test 7-10
7.3 Bus Protocol 7-10
7.3.1 Bus Arbitration 7-10
7.3.2 Bus Error 7-11
7.3.3 Wait States 7-11
7.4 Clipper Bus Operations 7-12
7.4.1 Read Operation 7-12
7.4.1.1 Bus Watch During Read Operations 7-13
7.4.2 Write Operation 7-14
7.4.2.1 Bus Watch During Write Operations 7-15
7.4.2.2 Global Write Operation 7-18
7.4.3 Read-Modify-Write Operation 7-19

7.4.4 Special Bus Operations 7-19
7.4.5 Interrupts 7-20

Chapter 8. HARDWARE DESIGN CONSIDERATIONS

8.1 Introduction 8-1
8.1.1 Memory Subsystem 8-2
8.1.2 I/O Subsystem 8-2
8.1.3 CLIPPER Bus Control and Interrupt Logic 8-2
8.2 Memory Subsystem 8-3
8.2.1 Memory Interface Operation 8-3
8.2.1.1 Bus Watch Support 8-3
8.2.1.2 Clear Dirty Support 8-4
8.2.1.3 Memory Errors 8-4
8.2.2 Boot Memory 8-4
8.3 I/O Subsystem 8-4
8.3.1 IOPs 8-5
8.3.2 Interbus Interface 8-6
8.3.3 Interbus Arbitration 8-6
8.3.4 Deadlock 8-7
8.4 Bus Control and Interrupt Logic 8-7
8.4.1 Interrupt Controller 8-7
8.4.1.1 Interrupt Controller Operation 8-8
8.4.1.2 Interrupt Controller Timing Considerations 8-9
8.4.1.3 Interrupt Controller Timing Example 8-10
8.4.2 CLIPPER Bus Arbiter 8-11
8.4.2.1 Bus Arbiter Operation 8-12
8.4.2.2 Bus Arbitration Timing Considerations 8-12
8.4.2.3 Bus Arbitration Timing Example 8-12
8.4.3 CLIPPER Module Interfacing 8-13

Chapter 9. SOFTWARE DESIGN

9.1 Introduction 9-1
9.1.1 The Bootstrap Program 9-1
9.1.2 The Monitor 9-2
9.1.3 The User Program 9-2
9.1.4 How the Code Is Presented 9-2
9.1.5 CLIPPER Register Definitions 9-4
9.2 Example CLIPPER System 9-16
9.2.1 I/O Configuration 9-16
9.2.2 Memory Configuration 9-16
9.2.3 Bootstrap Program Flow 9-16
9.2.4 Monitor Program Flow 9-17
9.3 An Example Bootstrap Program 9-18
9.3.1 Bootstrap Prologue—HTLB Pages 9-18
9.3.2 Bootstrap Initialization 9-20
9.3.3 Bootstrap Auxilliary Routines 9-23
9.4 An Example Monitor Program 9-26
9.4.1 Monitor Prologue 9-26
9.4.2 Memory Management Tables 9-34
9.4.3 Virtual Memory Tables 9-36
9.4.4 Initializing the Exception Vector Table 9-38
9.4.5 Final Initialization 9-38
9.4.6 The Command Loop 9-40
9.4.7 Table Initialization Routines 9-48
9.4.8 Miscellaneous Support Routines 9-66
9.4.9 Memory Management Routines 9-70
9.4.10 User Virtual Memory Support 9-86
9.4.11 Fatal Error Handlers 9-102
9.4.12 Trap Handlers 9-112
9.4.13 System Call Handlers 9-136
9.4.14 Interrupt Handlers 9-144
9.4.15 Exception Handler Support Routines 9-148
9.4.16 Monitor Data Segment 9-156
9.5 An Example User Program 9-160
9.5.1 Memory Usage 9-161
9.5.2 Prologue 9-162
9.5.3 The Main Body 9-164
9.5.4 SignOn—Print A Sign-on Message 9-166
9.5.5 CalcFibi—Calculate Fibonacci Sequence Iteratively 9-168
9.5.6 CalcFibr—Calculate Fibonacci Sequence Recursively 9-170
9.5.7 FibR—Calculate Fibonacci Number Recursively 9-172
9.5.8 PrintET—Print Elapsed Time Message 9-174
9.5.9 ConvertDA—Convert Value to Decimal ASCII 9-176
9.5.10 User Data Segment 9-178
9.6 Cross Reference Listings 9-180
9.6.1 Monitor Cross Reference 9-180
9.6.2 User Program Cross Reference 9-189

Chapter 10. INSTRUCTION SET

Appendix A. CLIPPER MODULE SIGNAL DESCRIPTIONS

Appendix B. INSTRUCTION SET SUMMARY

Appendix C. QUESTIONS AND ANSWERS

Index

LIST OF FIGURES

Figure 1.1 Computer Architectures 1-1
Figure 1.2 The Architectural Performance Gap 1-2
Figure 1.3 Partial List of Manufacturers Committed to UNIX System V 1-3
Figure 1.4 CLIPPER Module Block Diagram 1-4
Figure 1.5 Functional Integration Strategies 1-7
Figure 1.6 System Based on CLIPPER Microprocessor Module 1-10
Figure 2.1 CLIPPER Register Sets 2-3
Figure 2.2 Program Status Word 2-4
Figure 2.3 System Status Word 2-5
Figure 2.4 CLIPPER Primitive Data Types 2-7
Figure 2.5 Alignment of Data in Memory 2-8
Figure 2.6 Instruction Formats 2-9
Figure 2.7 Addressing Modes 2-11
Figure 2.8 Addressing Modes and Structured Data Types 2-15
Figure 2.9 Exception Vector Table 2-16
Figure 2.10 Supervisor Stack After INTRAP 2-18
Figure 2.11 Calculation of Exception Vector Table Entry 2-18
Figure 3.1 CLIPPER Memory Architecture 3-2
Figure 3.2 Main Memory Translations 3-3
Figure 3.3 A Simple Mapping Scheme 3-3
Figure 3.4 Page Based Memory 3-4
Figure 3.5 Two-Level Address Translation 3-5
Figure 3.6 Page Table Directory Format 3-6
Figure 3.7 CLIPPER Address Translation 3-6
Figure 3.8 Page Table Entry Format 3-7
Figure 3.9 HTLB Mapping 3-9
Figure 3.10 Demand Paging 3-10
Figure 3.11 CAMMU Block Diagram 3-11
Figure 3.12 Basic CAMMU Functional Flow 3-12
Figure 3.13 CLIPPER TLB 3-13
Figure 3.14 TLB Line Format 3-14
Figure 3.15 CPU Virtual Address Format 3-15
Figure 3.16 DTU Virtual Address Translation 3-15
Figure 3.17 CAMMU Control Register 3-16
Figure 3.18 CAMMU Reset Register 3-17
Figure 3.19 CAMMU Addressing 3-18
Figure 3.20 TLB Access Data Formats 3-19
Figure 3.21 PDO Register Access Format 3-20
Figure 4.1 Memory Hierarchy 4-1
Figure 4.2 CLIPPER Cache Access Time 4-2
Figure 4.3 Set Associativity 4-3
Figure 4.4 CLIPPER Cache 4-4
Figure 4.5 Cache Hit Rate 4-5
Figure 4.6 Cache Replacement Time 4-5
Figure 4.7 Cache Write Strategies 4-6
Figure 4.8 Bus Watch 4-7
Figure 4.9 CLIPPER Cache Access Mechanism 4-8
Figure 4.10 CLIPPER Cache Line Format 4-9
Figure 5.1 Pipelining 5-1
Figure 5.2 CLIPPER Pipeline 5-2
Figure 5.3 CLIPPER CPU Block Diagram 5-3
Figure 6.1 FPU Programming Model 6-2
Figure 6.2 Floating-Point Formats 6-7
Figure 6.3 Approximate Real Range for Single-Precision 6-8
Figure 6.4 Approximate Real Range for Double-Precision 6-8
Figure 6.5 IEEE 754-Bit Encodings for Single- and Double-Precision Values 6-9
Figure 7.1 CLIPPER Bus Signals 7-1
Figure 7.2 CLIPPER Bus 7-2
Figure 7.3 Reset Timing 7-9
Figure 7.4 Single Word Read 7-11
Figure 7.5 Single Word Read Flowchart 7-12
Figure 7.6 Quadword Read 7-12
Figure 7.7 Quadword Read Flowchart 7-13
Figure 7.8 Watch I/O Reads 7-13
Figure 7.9 Single Word Write 7-14
Figure 7.10 Single Word Write Flowchart 7-15
Figure 7.11 Quadword Write 7-15
Figure 7.12 Quadword Write Flowchart 7-16
Figure 7.13 Quadword Write 7-16
Figure 7.14 Watch CPU and I/O Writes 7-17
Figure 7.15 Read-Modify-Write 7-17
Figure 7.16 Read-Modify-Write Flowchart 7-18
Figure 7.17 D-CAMMU Read from Companion I-CAMMU 7-18
Figure 7.18 D-CAMMU Write to Companion I-CAMMU 7-19
Figure 7.19 DTU Operation 7-20
Figure 7.20 Cache Line Replacement 7-21
Figure 7.21 Bus Retry 7-21
Figure 7.22 Bus Error 7-22
Figure 7.23 $\overline{\text{IRQ}}$ Timing 7-23
Figure 7.24 $\overline{\text{NMI}}$ Timing 7-23
Figure 8.1 CLIPPER System 8-1
Figure 8.2 Traditional I/O Implementation 8-5
Figure 8.3 IOP 8-6
Figure 8.4 Interrupt Controller 8-8
Figure 8.5 Interrupt Controller Timing Example 8-10

Figure 8.6 Bus Arbiter 8-11
Figure 8.7 Bus Arbiter Timing Example 8-13
Figure 8.8 CLIPPER System (Block Level) 8-14
Figure 8.9 CLIPPER Module Buffering 8-15
Figure 9.1 Supervisor Stack in CSwitch 9-46
Figure 9.2 Master Initializer Table Format 9-48
Figure 9.3 Monitor Virtual Memory Map 9-56
Figure 9.4 Monitor Virtual Memory Usage 9-58
Figure 9.5 Initial Real Memory Usage 9-59
Figure 9.6 Vector Table—Traps 9-62
Figure 9.7 Vector Table—System Calls 9-64
Figure 9.8 Vector Table—Interrupts 9-64
Figure 9.9 Virtual to Real Address Translation 9-80
Figure 9.10 Virtual to Real Address Calculation 9-81
Figure 9.11 Initial User Virtual Memory Usage 9-86
Figure 9.12 Window Map During LoadPage Call 9-94
Figure 9.13 Inital User Virtual Memory Map 9-100
Figure 9.14 Real Memory Usage After User Program Loaded 9-101
Figure 9.15 Supervisor Stack Within an Exception Handler 9-106
Figure 9.16 PutHexW R0 Contents 9-108
Figure 9.17 Final User
Figure 9.17 Virtual Memory Usage 9-161
Figure 9.18 Final User Virtual Memory Map 9-161
Figure 9.19 Real Memory Usage When User Program Exits 9-162
Figure 9.20 Stack Upon Entry to FibR 9-170
Figure 10.1 Instruction Description Example 10-3
Figure C.1 Instruction Buffer C-3
Figure C.2 Load Byte Operation C-9
Figure C.3 Store Byte Operation C-9

LIST OF TABLES

Table 2.1 Conditional and Unconditional Traps 2-19
Table 2.2 Trap Handler Environment Summary 2-19
Table 3.1 Page Access Encoding 3-8
Table 3.2 HTLB Virtual Address Translations 3-13
Table 6.1 IEEE Required Operations 6-10
Table 6.2 Infinity Arithmetic 6-11
Table 7.1 Signal Summary 7-3
Table 9.1 Fibonacci Sequence 9-160
Table 9.2 Fibonacci Formal Definition 9-160
Table 10.1 Alphabetical Listing of the Instructions 10-2
Table 10.2 Syntax Field Operands 10-4
Table 10.3 Operation Field Operands 10-4
Table 10.4 Operation Field Operators 10-5
Table 10.5 Format Field Operands 10-5
Table 10.6 Example Field Operands 10-6
Table 10.7 Integer Branch Conditions 10-16
Table 10.8 Floating Branch Conditions 10-17
Table B.1 Instruction Set By Function B-1
Table B.2 Instruction Opcode/ Mnemonic Summary B-6
Table B.3 Macro Instruction Code Field (Opcode B4) B-7
Table B.4 Privileged Macro Instruction Code Field (Opcode B6) B-7
Table C.1 Instruction Buffer Pipelining C-2

Preface

The CLIPPER microprocessor is an advanced, high-performance 32-bit computer architecture, drawing on mainframe/supercomputer techniques, and implemented in 3 CMOS VLSI chips. The CLIPPER chip set consists of a CPU, containing an IEEE Standard Floating-Point Unit, and two combination cache-plus-memory management units (CAMMUs), one for data and one for instructions. The processor makes extensive use of concurrency, including a sophisticated pipeline and separate buses for instructions and data. The CLIPPER employs a load-store instruction architecture and implements its instructions with hardwired logic instead of microcode. Consequently, most frequently used instructions execute in one 30 ns clock cycle.

This document is a comprehensive description of the CLIPPER architecture. All important architectural concepts are covered, along with a detailed technical explanation including many examples. The book is intended for both hardware and software designers using the CLIPPER microprocessor, engineers who are simply curious about CLIPPER, and students learning about state-of-the-art computer architectures.

Chapter 1 provides a high-level overview of computer architectural concepts and how they influenced the design of the CLIPPER microprocessor. Chapters 2–6, 9, and 10 will be of particular interest to software engineers. Chapters 2–5 cover the programming model, memory management, caches, and the CLIPPER instruction pipeline. Chapter 6 describes the floating-point execution unit located on the CPU/FPU chip and provides programming examples. Chapter 9 provides additional software design examples. In Chapter 10 the CLIPPER instructions are described individually.

Chapters 7 and 8 are intended for hardware designers. Chapter 7 describes the CLIPPER bus and chapter 8 provides hardware design guideline examples.

For more information on the CLIPPER, consult the following documents:

Introduction to the CLIPPER Architecture. A summary of the key architectural features of the CLIPPER. Available now.

CLIPPER 32-Bit Microprocessor Product Description. A reference document for the CLIPPER Module. Available now.

CLIPPER Cross Support Package User's Manual. A cross development manual for the compiler, simulator, debugger and other development tools that comprise the cross support package. Available now.

CLIX System V Manual Set. A set of six binders containing the manuals for the CLIX operating system (CLIPPER operating sytem derived from UNIX System V). Available 3rd Quarter 1986.

CLIPPER Systems Software Writer's Guide. A detailed guide to writing operating systems and optimizing compilers. Available 1st Quarter 1987.

CLIPPER Hardware Reference Manual. A detailed guide to hardware interfacing to the CLIPPER bus. Available 4th Quarter 1986.

CLIPPER C Compiler Manual. Describes the implementation of the C programming language for the CLIPPER microprocessor. Available 3rd Quarter of 1986.

CLIPPER FORTRAN Compiler Manual. Describes the implementation of the FORTRAN programming language for the CLIPPER microprocessor. Available 3rd Quarter of 1986.

CLIPPER Pascal Compiler Manual. Describes the implementation of the Pascal programming language for the CLIPPER microprocessor. Available 3rd Quarter of 1986.

Chapter 1
Overview

1.1 INTRODUCTION

The CLIPPER architecture brings supercomputer and mainframe technology to a 32-bit microprocessor. Its advanced architecture permits it to achieve an average rate in excess of the performance of a VAX 8600. Its built-in floating-point unit performs at greater than 1 million floating-point operations per second (MFLOPS). The CLIPPER has a 4 gigabyte (2^{32}) address space, and it includes hardware support for demand paged virtual memory.

The CLIPPER was designed for applications that demand high performance—engineering workstations, graphics subsystems, and the latest industrial automation products. It makes possible for the first time many computation and memory intensive applications, such as speech recognition and robot vision systems, in a cost-effective package. Until the CLIPPER became available, these projects required superminicomputers, mainframes, or even supercomputers.

Despite its many advanced features, the CLIPPER was designed to be compatible with existing software. By providing optimizing compilers targeted to the CLIPPER for several standard programming languages, and a CLIPPER implementation derived from the UNIX System V standard operating system, Fairchild has ensured that most existing programs can easily be moved to the CLIPPER. Fairchild also offers a complete set of software development utilities for the CLIPPER, including interactive debuggers and simulators.

The following sections of this chapter discuss the innovative features of the CLIPPER architecture, including

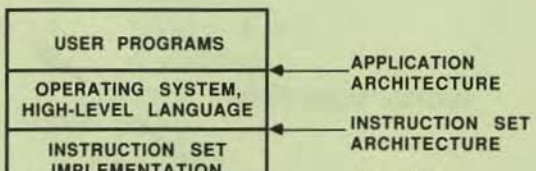
- Supercomputer and mainframe derived features, such as dual internal buses, advanced cache and pipeline designs, and concurrent processing units.
- The CLIPPER's streamlined instruction set (many instructions execute in one clock cycle—30 ns).
- Hardware support for virtual memory and other operating system functions.
- The architecture of CLIPPER-based systems, including the CLIPPER module itself, the I/O subsystem, and the memory subsystem.
- The high-level language and operating system "architecture" of the CLIPPER.

But first, we will explain exactly what we mean by computer architecture, and why the world needs another one.

1.2 WHY A NEW ARCHITECTURE?

The term *computer architecture* is often used as if it meant simply "the organization and design of a computer". In fact, however, there are several distinct architectures within a computer system, each defined by the boundary between different levels of the system. Figure 1.1 shows an abstract picture of a computer system, with the simplest operations and functions on the bottom and the most complex user-dependent operations on the top. Each level makes use of the functions provided by the level below. For simplicity,

Figure 1.1 Computer Architectures



we have shown only four levels, and thus three architectures.

The *application architecture* is the interface between the application program and the high-level programming language plus operating system. Most application programs are written in standard programming languages, such as C, Pascal, or Fortran, and most are designed to execute on standard operating systems, such as UNIX. The language specification and OS system calls define the application architecture.

The *instruction set architecture* is the interface between assembly language programs and the computer hardware. It is defined by the registers, instruction formats, and addressing modes of the processor.

The *internal architecture* is defined by the gates, buses, caches, and execution units of the processor. This architecture is normally invisible to user-supplied software, but it has a great bearing on performance, as we shall see.

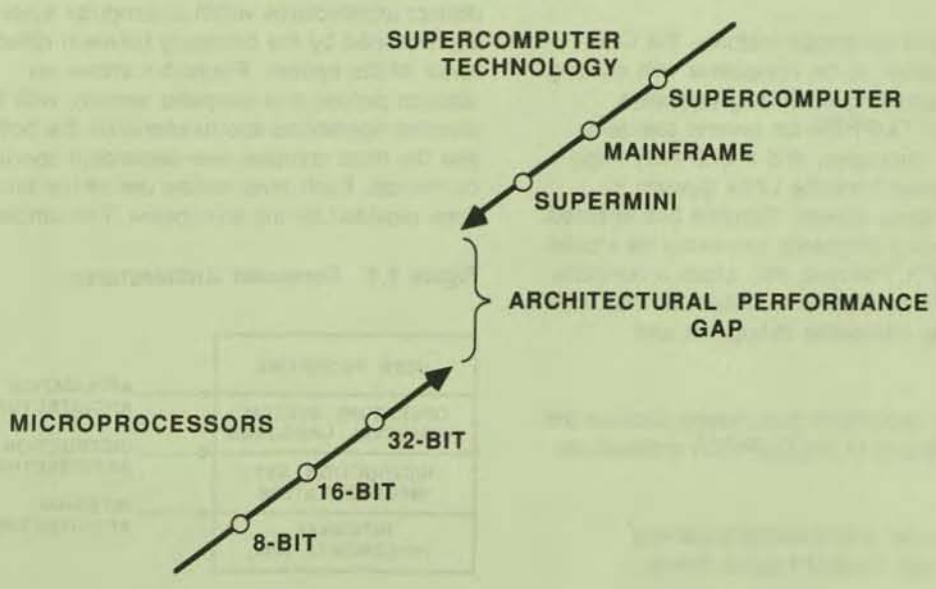
The CLIPPER has new internal and instruction set architectures, although it preserves a standard application architecture. These new low-level architectures incorporate technologies derived from supercomputers and mainframes, technologies that have never been applied to microprocessors before. Because of its new architecture, the CLIPPER has achieved a quantum leap in performance over other microprocessors—and that is the answer to "why another architecture?"

1.2.1 The Architectural Performance Gap

The advanced features of the CLIPPER architecture are not new to the world of computers in general, they are only new to the world of microprocessors. This fact explains why there has been a performance gap between microprocessors and large computers, such as mainframes and superminicomputers, even when the microprocessors ran at the same clock rate as the big machines and had the same word size.

Figure 1.2 shows a schematic representation of computer evolution. The lower, ascending line

Figure 1.2 The Architectural Performance Gap



represents the increasing power of microprocessors from generation to generation, with the latest 32-bit microprocessors at the high-end. The upper, descending line represents the migration of supercomputer technology down from machines such as the Cray to mainframes and superminicomputers. The *architectural performance gap* in the middle is the result of microcomputer designers failing to incorporate advanced architectural features. It is this gap that the CLIPPER has jumped. In fact, in many ways the CLIPPER is a low-end supercomputer rather than a high-end microprocessor.

When microprocessors were first developed, the sheer feat of getting a computer on a chip was so impressive that it almost seemed uncharitable to ask for more. And the severe size limitations of these early processors precluded many fancy techniques. But the latest generation chips contain almost half a million transistors, and there are no longer any reasons for not employing the latest architectural technology as well for high performance processors.

1.2.2 Compatibility—Preserving Your Software Investment

Since an increasing portion of the value of computer-based systems is in the software investment, preserving this investment has become a central concern of system implementers. The CLIPPER was designed to insure that users would get the best of two worlds, i.e., reap the performance benefits of supercomputer technology while preserving their existing software investment.

Until recently, this was not possible. Designers were basically faced with the choice of keeping their software, with only marginal performance gains possible, or moving to a new generation in performance, but having to scrap the existing software. The reason for this painful choice was that truly compatible computers had to be *instruction set compatible*, that is, their instruction set architectures had to be the same.

Instruction set compatibility (often called *binary compatibility* because the binary versions of

Figure 1.3 Partial List of Manufacturers Committed to UNIX System V

AT&T Information Systems	Durango Systems	Perkin-Elmer Corporation
Alpha Micro	Elxsi	Plexus Computers Inc.
Altos Computer Systems	Encore Computer Corporation	Prime Computer, Inc.
Amdahl Corporation	Fairchild Semiconductor Corporation	Pyramid Technology Corp.
Apollo Computers Inc.	Flexible Computer Corporation	Ridge Computers
Arete	Gould Inc.	Sequent Computer Systems, Inc.
BBN Computer	Harris Corporation	Siemens AG
Bull	Heurikon Corp.	Silicon Graphics
Callan Data	Hewlett-Packard	Sperry Corporation
Celerity Computing	Honeywell Information Systems, Inc.	Sun Microsystems, Inc.
Centurion	IBM Corp.	Tandy Corporation
Charles River Data Systems	ICL	Tektronix, Inc.
Control Data Corporation	Intel Corp.	Texas Instruments
Convergent Technologies	Intergraph Corporation	Three Rivers
Convex Computer Corporation	Lanier	Tolerant Systems, Inc.
Cray Research Inc.	Motorola/Four Phase Systems	Toshiba
Cromemco	NCR Corporation	Valid Logic
Daisy Systems	National Semiconductor Corporation	Wicat
Data General	Nixdorf	Zilog, Inc.
Digital Equipment Corp.	Olivetti	

application programs will run on a compatible instruction set) was important for two reasons: (1) most programs then were written in assembly language, and (2) no standard operating system existed. In recent years this situation has changed drastically. Now most application and system programs are written in a *high-level language*, and *Unix System V* has emerged as the *de facto* standard operating system, available now on dozens of different computer architectures. Figure 1.3 lists some of the manufacturers committed to Unix System V.

As a result of this change computer manufacturers have been freed from the crippling constraint of instruction set compatibility with antiquated architectures. The compatibility that counts now is *application architecture* compatibility. And the user gets the benefit, because much larger performance gains can be achieved that preserve compatibility at this level.

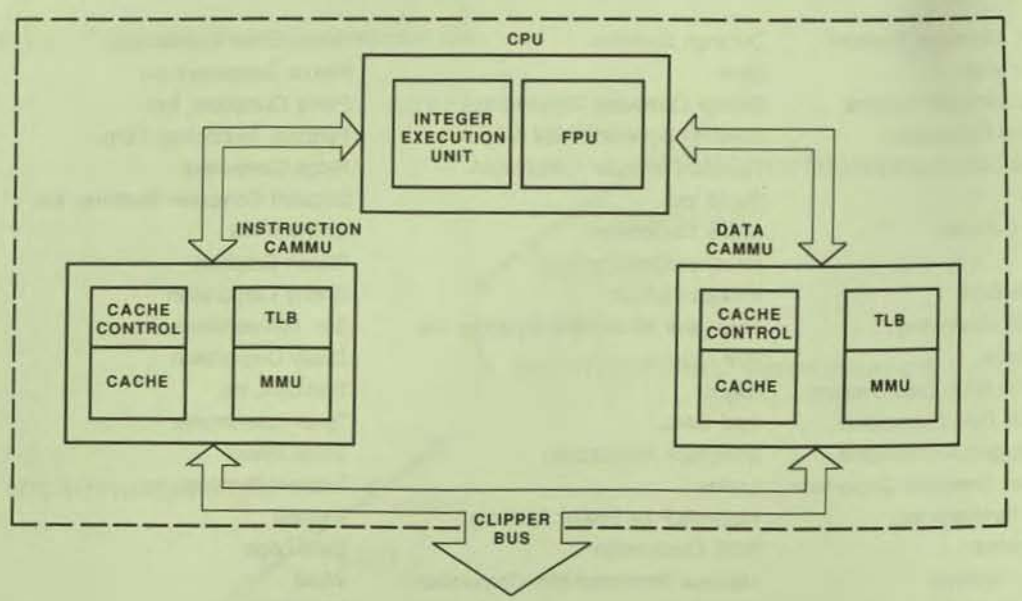
1.3 HIGH PERFORMANCE

The high performance of the CLIPPER, as with mainframes and supercomputers, is the result of a balance among several factors and of careful tuning; it is not the result of one all-powerful feature. The most obvious starting point is the clock speed: the CLIPPER runs at a clock rate of 33.3 ns, faster than any other 32-bit microprocessors. But in order to utilize that clock rate, the architects of the CLIPPER had to carefully balance and tune four other key elements of the internal architecture:

1. Caches.
2. Internal dual buses.
3. Pipelines.
4. Integrated execution units.

In addition, the instruction set is designed to use simple, register-to-register instructions, the most common of which execute in one clock cycle, and

Figure 1.4 CLIPPER Module Block Diagram



special hardware support is provided for critical operating system functions, such as virtual memory. The next four subsections will cover the four elements of the CLIPPER's internal architecture that contribute to performance, while the remaining sections will discuss features of the instruction set and of the OS support mechanisms.

Figure 1.4 shows a block diagram of the CLIPPER module. The module contains a CPU chip and two Cache-MMU chips (called *CAMMUs*). One CAMMU is dedicated to instructions, the other to data, and each is connected to the CPU by its own dedicated bus. The CPU contains an on-chip Floating-Point Unit (FPU). We will refer repeatedly to this diagram in the following subsections.

1.3.1 Advanced Cache Management

Caching is basically a cost effective way of creating a computer system that looks like it has a very fast (and very expensive) main memory—but doesn't. The cache is a small, fast buffer with an access time that is intermediate (60-120 ns) between CPU registers (30 ns) and the low speed, inexpensive dynamic RAMS (150 ns) that are actually used to build the main memory. The caching hardware fills this buffer with recently accessed instructions and data; then at some later time, the processor often finds the data or instruction it wants in the cache and doesn't have to go all the way to main memory. (This occurs because programs often exhibit *locality*, that is they tend to access the same address after a short time interval—temporal locality, or else nearby addresses—spatial locality.) The *effective access time* is thus usually much lower than for main memory, although the actual number depends on the *cache hit rate*, which in turn depends on the cache design and the actual instruction mix.

If cost is no obstacle, main memory can be simply built out of 30 ns chips, and this is exactly how supercomputers are designed. Consequently, many supercomputers do not use caches. Mainframe computers, on the other hand, make extensive use of caching, and the CLIPPER's advanced caching mechanisms are largely derived from the experience of mainframes. The details of the cache architecture are covered in Chapter 4; we will simply summarize the facts here.

The CLIPPER Module contains two Cache/MMU chips (see Figure 1.4). Each CAMMU contains three caches:

- a 4K byte, 2-way associative data or instruction cache, with a 16-byte line length (these terms are covered in detail in Chapter 4)
- a quadword (16-byte) line buffer, that acts as a superfast cache-within-the-cache
- a Translation Lookaside Buffer (TLB) that caches 128 frequently used virtual-to-real address translations

When evaluating cache performance, one key metric is the *hit rate*, the probability that the requested item is found in the cache instead of slower speed memory. Another key figure is the *CPU access time*, the time the CPU requires to access an item. CPU access time includes the raw access time of the memory or cache chip plus the transfer time over the bus. For example, as we said above, the raw access time of a DRAM chip is about 150 ns, but the actual CPU access time to main memory is around 500 ns, because of bus transfer times and the resulting wait states.

The cache has a hit rate of greater than 90% and a CPU access time on a cache hit of 120 ns. The line buffer has a hit rate of 58% (instructions) or 41% (data) and a CPU access time of 60 ns (hit). The TLB has a hit rate of over 99% and a CPU access time for a hit of 60 or 120 ns. The result of the complex interplay of the three caches is an effective CPU access time to memory of 100 ns (instructions) or 150 ns (data). This means that the CLIPPER behaves as if its main memory were made up of high-speed static RAMs, and the bus had no wait states, even though it really uses inexpensive DRAMs and a less complex bus.

In order to implement this sophisticated caching system, the CLIPPER employs many advanced mechanisms, including:

- Bus Watch: a mechanism for guaranteeing data consistency with copy-back caching (see Chapter 4).
- Simultaneous access to the TLB for the physical address and to the main cache for instructions or data at that address; thus there is no extra delay for address translation (see Chapter 3).
- Burst memory transfers (see below).

1.3.2 Increasing Bus Bandwidth

The factor that limits the performance of most microprocessors is *bus bandwidth*, the amount of data per unit of time that can be transferred over the key buses in the system. One of the most distinctive features of the CLIPPER internal architecture is the use of two buses between the CPU and the CAMMUs; one bus is used for data, the other for instructions. The use of two buses effectively doubles the bus bandwidth in this critical region. The use of two buses and two CAMMUS makes the caches more efficient as well, because locality is much stronger for pure instructions and pure data than for a mixture of the two.

The CLIPPER also maximizes the bandwidth of the CLIPPER bus between the Module and main memory. Through the use of burst mode transfers, the CLIPPER is able to send 16 data bytes (4 words) for each address word. Conventional buses transfer one word of data for each address word.

1.3.3 Sophisticated Pipelining

The CLIPPER has a pipelining system that is very similar to that of supercomputers. It consists of a pipeline with three phases:

1. fetch
2. decode
3. execute

(Since the CLIPPER is a load/store machine, it doesn't need a separate address calculation stage or a separate address pipeline. The address in a load or store is simply calculated during the execute phase.)

The CLIPPER pipeline also has some significant novelties:

- The execution unit itself is partitioned into *three stages*, which execute in parallel and can overlap for successive instructions. Each stage takes only one clock cycle to complete, so the effective throughput is typically one instruction per clock. This pipelined execution unit is sometimes referred to as the *lower pipeline*, while the fetch and decode phases are referred to as the *upper pipeline*.
- The CLIPPER includes a hardware resource manager that handles resource contention automatically. (In supercomputer terminology, this is called *scoreboarding*.)
- The execution unit includes a feedback mechanism so that intermediate results from one calculation are immediately available as input for the next calculation.
- The instruction CAMMU prefetches instructions into the CAMMU from main memory. It is able to do this because a copy of the Program Counter is maintained on the CAMMU.

1.3.4 Integrated Processing Units

One reason for the CLIPPER's spectacular performance is its high level of integration. The CLIPPER Module contains just three chips, instead of the dozens of chips that would be required to duplicate its functionality in more conventional microprocessor systems. The CAMMU alone replaces more than 30 chips in conventional cache designs. But beyond simply increasing the level of integration, the CLIPPER architecture was carefully partitioned to separate functions that should be integrated onto one chip from functions that should be on separate chips (given the current state of the art in VLSI).

An example of this careful partitioning is in the relationship among the CLIPPER's CPU, FPU, Cache, and MMU. Of course eventually all these functions will be integrated onto one chip; but that will require almost a million transistors, and VLSI technology is still a few years away from that level. For the present, more than one chip is required; so the question is, which functions should be grouped together and which should be separated?

The traditional approach is to combine the CPU and MMU, and leave the FPU and cache as separate chips (actually, in the traditional approach the user implements the cache). Figure 1.5 shows a diagram of this approach. This strategy has two disadvantages:

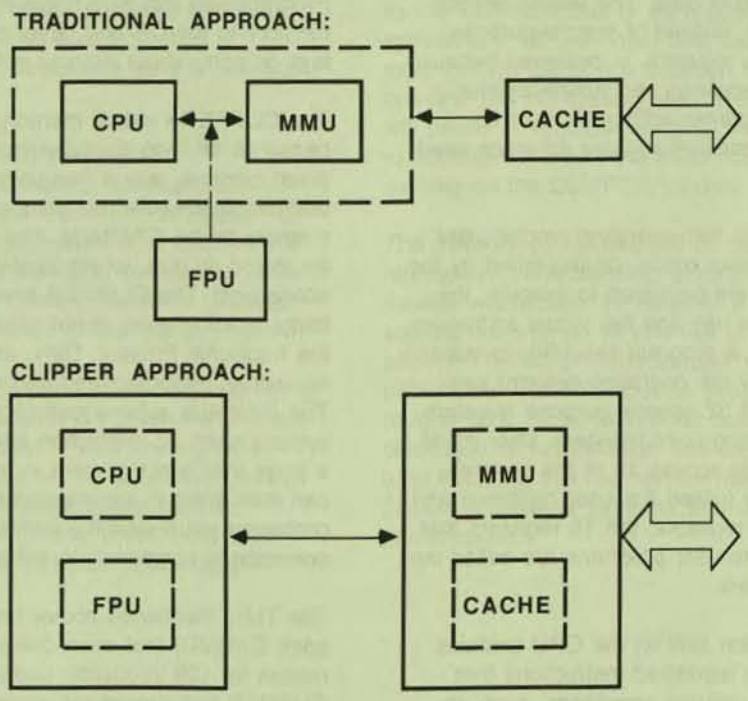
- Even though the MMU stands logically *between* the FPU and the cache (as it does between the CPU and the cache), the FPU has no direct access to the MMU, and the CPU is burdened with an overhead for every FPU address translation.
- Since the cache is left out of the integration picture altogether, no benefit can be derived from overlapping memory translations with cache accesses.

The CLIPPER architecture, on the other hand, integrates all computational functions onto one chip (the CPU) and all memory access functions onto another (the CAMMU). The CPU chip contains both an integer execution unit and an on-board floating-point unit and the CAMMU contains both an MMU and a cache memory. On each chip the units can execute concurrently. Floating-point calculations can be done at the same time integer instructions are executing, and address translation occurs concurrently with cache lookup.

1.4 STREAMLINED INSTRUCTION SET

The CLIPPER's instruction set was designed for rapid execution of compiled code, especially code produced by compilers that optimize register usage. All unnecessary operations that are hard for compilers to use have been eliminated; the operations that remain are implemented with

Figure 1.5 Functional Integration Strategies



hardwired logic, not microcode, and many execute in one clock cycle. In many ways, the CLIPPER's instruction set resembles the scalar instruction set of a supercomputer.

All arithmetic and logical instructions manipulate data in registers; only loads, stores, calls, and branches access memory directly. Load and store instructions are supplied with a set of nine addressing modes for efficient access to the elements of typical high-level language data structures, such as arrays, records, and arrays of records. All addresses are unsegmented 32-bit quantities; thus the logical address space and the physical address space are both 4 gigabytes.

The CLIPPER architecture has a 32-bit Program Counter and thirty-two 32-bit storage registers, more than any other commercial microprocessor. In addition, the CLIPPER contains eight 64-bit storage registers that are dedicated to floating-point arithmetic. The 32-bit registers are general purpose; each one can be used to hold either an address or a word of data. The use of general-purpose registers, instead of special-purpose data and address registers, is preferred because it eliminates unnecessary register-to-register transfers when address arithmetic must be performed and simplifies register schemes used by compilers.

The CLIPPER has two *operating modes*, user mode and supervisor mode, distinguished by the instructions they are permitted to execute, the registers they can use and the virtual addresses they can access. A program executing in supervisor mode (usually the operating system) can access data in all 32 general-purpose registers and all eight floating-point registers. User mode programs can only access 16 of the general-purpose registers (called the *user registers*) and the floating-point registers; the 16 registers that are inaccessible to user programs are called the *supervisor registers*.

A *Macro Instruction Unit* on the CPU contains sequences of the hardwired instructions that implement more complex operations, such as string manipulation and operating system support. The macro instructions access their own private registers and need not use general purpose registers, so their execution places no additional burden on programs. The combination of the load/store instruction set plus the Macro Instruction Unit gives the CLIPPER many of the advantages of both a reduced instruction set computer (RISC) and a complex instruction set computer (CISC).

1.5 VIRTUAL MEMORY AND OPERATING SYSTEM MECHANISMS

The CLIPPER provides hardware support for demand paged virtual memory, plus support for fast operating system calls and context swaps.

In many computer systems the logical address space is far larger than the actual main memory hardware. *Virtual Memory* is a mechanism implemented by the operating system (usually with hardware support) for circumventing the limits on physical memory size. Under a virtual memory system, it appears to users as if the entire logical address space were available for storage. Programmers can thus leave management of memory to the OS and never have to write overlays or worry about running out of memory.

The CLIPPER's virtual memory mechanisms are based on 4K byte pages of memory. At any given moment, only a few pages of the logical address space need be mapped into physical memory by the CAMMUs. The other pages can be stored on disk, where cost-per-bit is more economical. The CLIPPER keeps track of several items of information about each page, including the traditional Present, Dirty, and Accessed bits, as well as protection and caching specifications. The CAMMUs automatically trap to the operating system when an instruction attempts to reference a page that is not present in memory; the OS can then swap in the missing page, and the processor will restart the instruction. This whole operation is completely invisible to the end user.

The TLBs mentioned above are caches (one on each CAMMU) that each contain mapping information for 128 frequently used pages. Thus the CLIPPER has immediate access to 256 pages or one million bytes of memory and therefore can translate most addresses without ever having to

consult a memory-based page table (see Chapter 3 for more details on mapping). The typical hit rate for the TLB is greater than 99% for a single process, which may seem like overkill, but the access time for a TLB miss (having to look up an address in a page table) is at least 10 times the access time for a TLB hit. So going from a 98% hit rate to a 99% rate yields an 8% performance improvement. It's all part of the CLIPPER's performance tuning.

The other operating system support mechanisms are designed to make possible fast system calls. The separate register files for system and user mean that less context loading and storing is required when a system call is made, thus significantly reducing the system overhead. Exception processing is handled in the Macro Instruction Unit, which makes use of its own private registers instead of the system's registers. System calls can be vectored directly to the appropriate call handler instead of having to go through a common routine that calls the proper handler. Finally, the interrupt offset is passed on a separate interrupt bus, so no bus arbitration is required for external devices to send an interrupt vector.

1.6 SOFTWARE AND SUPPORT

Fairchild supplies a complete development environment to go with the CLIPPER. In addition, a high-performance implementation of System V is available. The CLIPPER's port makes use of the demand paging and caching hardware to implement a high performance virtual memory system.

The development environment includes optimizing compilers for C, Pascal, and Fortran, all of which can either be hosted on a VAX running either 4.2 BSD or ULTRIX or on the CLIPPER running System V. Along with the compilers are an assembler and linker, plus a VAX-hosted CLIPPER simulator, and an interactive debugger.

1.7 SYSTEM ARCHITECTURE

The CLIPPER Module is really an 846,000 transistor compute engine that happens to be implemented on three chips instead of one. But the module is supplied as a single unit, complete with clock logic, all on a small circuit board. The module plugs into a CPU motherboard. See Figure 1.6.

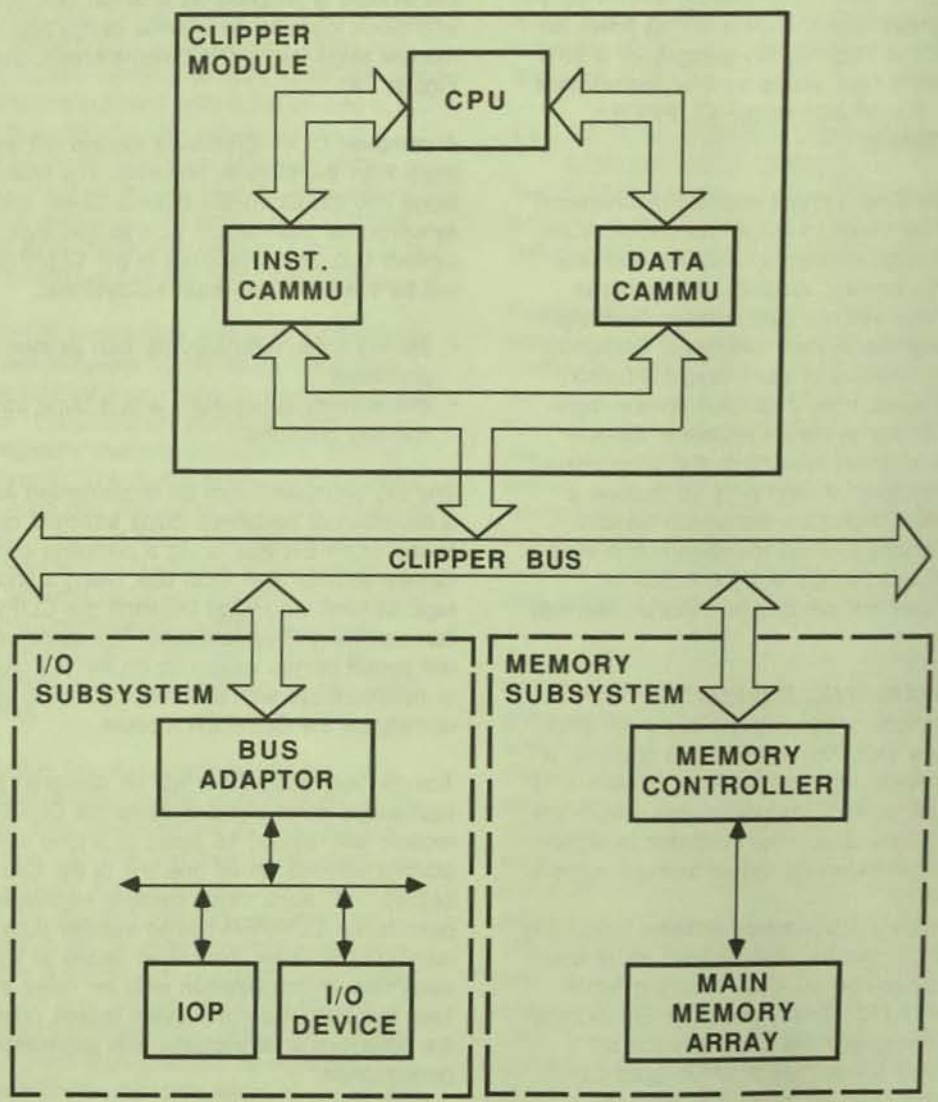
A complete CLIPPER-based system will include more than the Module, however. The Module plugs into the CLIPPER bus—a 32-bit, fully synchronous bus—which acts as the main local system bus. Also attaching to the CLIPPER bus will be the two other main subsystems:

- the I/O subsystem, usually with its own I/O processor
- the memory subsystem, with a burst transfer memory controller

The I/O subsystem can be implemented as either a collection of peripheral chips attached directly to the CLIPPER Bus or as a complete computer system with its own local bus, using standard logic to form the bridge between the CLIPPER bus and the I/O system bus. This standard logic will permit drivers executing on the I/O processor to communicate with the main operating system running on the CLIPPER module.

The memory subsystem will be designed for high-speed burst transfer, since the CLIPPER module will request 16 bytes at a time with one starting address (to fill one line of the CAMMU cache). This burst mode transfer mechanism permits the CLIPPER bus to transfer data roughly 50% faster than other buses at the same clock rate. In combination with the other architectural features, this mechanism makes possible the construction of systems with unparalleled performance.

Figure 1.6 System Based on CLIPPER Microprocessor Module



Chapter 2
CLIPPER Programming Model

2.1 INTRODUCTION

A primary design objective of the CLIPPER was to provide an instruction set architecture that is well suited for operating systems and compilers. In this context, "well suited" means that the architecture facilitates two somewhat divergent goals:

- The instructions should execute rapidly, preferably one per clock cycle.
- It should be easy to write programs using the instructions, especially operating systems, and easy for compiler code generators to target them.

The first goal pushes the designer towards a very simple instruction set, with few higher level concepts supported in the hardware. This approach is often called the Reduced Instruction Set Computer (RISC) philosophy. The second goal pushes the designer in quite a different direction—towards a richer instruction set with direct support for concepts from high-level languages and operating systems. This latter approach has several names—language directed architectures, high-level architectures, etc., depending on the concepts supported—but we will refer to them all as the Complex Instruction Set Computer (CISC) philosophy.

The CLIPPER instruction set is a balance between the RISC and CISC approaches. This balance is also a characteristic of supercomputer architectures, which contain both RISC and CISC features. Specifically, the CLIPPER instruction set architecture derives the following basic features from the RISC and CISC approaches:

RISC — Arithmetic and logical instructions operate on registers; basically only loads, stores, and branches, along with stack manipulation instructions access memory.

There are plenty of registers: thirty-two 32-bit registers, sixteen for the operating system and sixteen for the user.

The instruction format is designed to simplify decoding by making all instructions simple multiples of one basic size (16 bits), and the most frequently used instructions are shortest.

CISC — Instructions that do access memory are provided with a complete set of addressing modes that facilitate access to high-level data structures.

Explicit support for stacks and strings is provided.

Separate modes of operation are provided for the user and the operating system, each mode having its own resources and privileges.

Explicit support is provided for key operating system functions, such as system calls, exception handling, and virtual memory.

In this chapter, we will discuss the instruction set architecture of the CLIPPER in five sections: (1) registers, (2) data types and instructions, (3) addressing, (4) CLIPPER's two operating modes, and (5) exception handling.

2.2 REGISTER SETS

The CPU contains three sets of registers: 16 user registers (r0–r15), 16 supervisor registers (r0–r15) and 8 floating-point registers (f0–f7). The user and floating-point registers are accessible in both user and supervisor modes; the supervisor registers are accessible only in supervisor mode

(see Section 2.5 for a discussion of user and supervisor modes).

The CPU also has two status and control registers, the Program Status Word (PSW) and the System Status Word (SSW). The PSW is accessible in both user and supervisor modes; the SSW is writable only in supervisor mode.

The Program Counter (PC), also on the CPU, contains the address of the current instruction. It is accessible in both user and supervisor modes.

In addition to the CPU registers, each CAMMU has five registers which control various aspects of address translation, cache control, and fault recovery. These registers are accessible in supervisor mode only.

Figure 2.1 shows a diagram of all CLIPPER Module registers.

2.2.1 User And Supervisor Registers

The user and supervisor registers, r0–r15, are general-purpose, 32-bit data or address registers. They are used for word (32-bit) and longword (64-bit) integer operations.

Longword values are stored in register pairs, with the least significant word in the even-numbered register. Bytes and halfwords used in load and store instructions are sign- or zero-extended to 32 bits before they are put in registers.

2.2.2 Floating-Point Registers

The floating-point registers, f0–f7, are used by the FPU for floating-point instructions, which are executed concurrently with instructions in the integer Arithmetic Logic Unit (ALU). These 64-bit registers are used for floating-point operands in both single- and double-precision IEEE format. Single-precision operands stored in floating-point registers have zeros in the 29 least significant fraction bits and in the 3 most significant exponent bits.

The integer multiply and divide instructions are executed by the FPU, but use the general-purpose registers r0–r15 (user or supervisor) for these operations.

2.2.3 Program Status Word (PSW)

The PSW, shown in Figure 2.2, contains flags which identify and, together with the SSW, control a program's response to various exception conditions resulting from integer and floating-point operations. On hardware reset, the trace trap (T) flag is cleared; the remaining bits are undefined.

The PSW flags are summarized below. More detailed descriptions of the use of the CPU trap status (CTS), memory trap status (MTS), T, and the condition code flags are given in Section 2.6, Exceptions. The floating rounding mode (FR), enable floating trap (EFT), floating-point exception flags, and enable floating trap flags are described in more detail in Section 6.9.2, Floating-Point Exceptions.

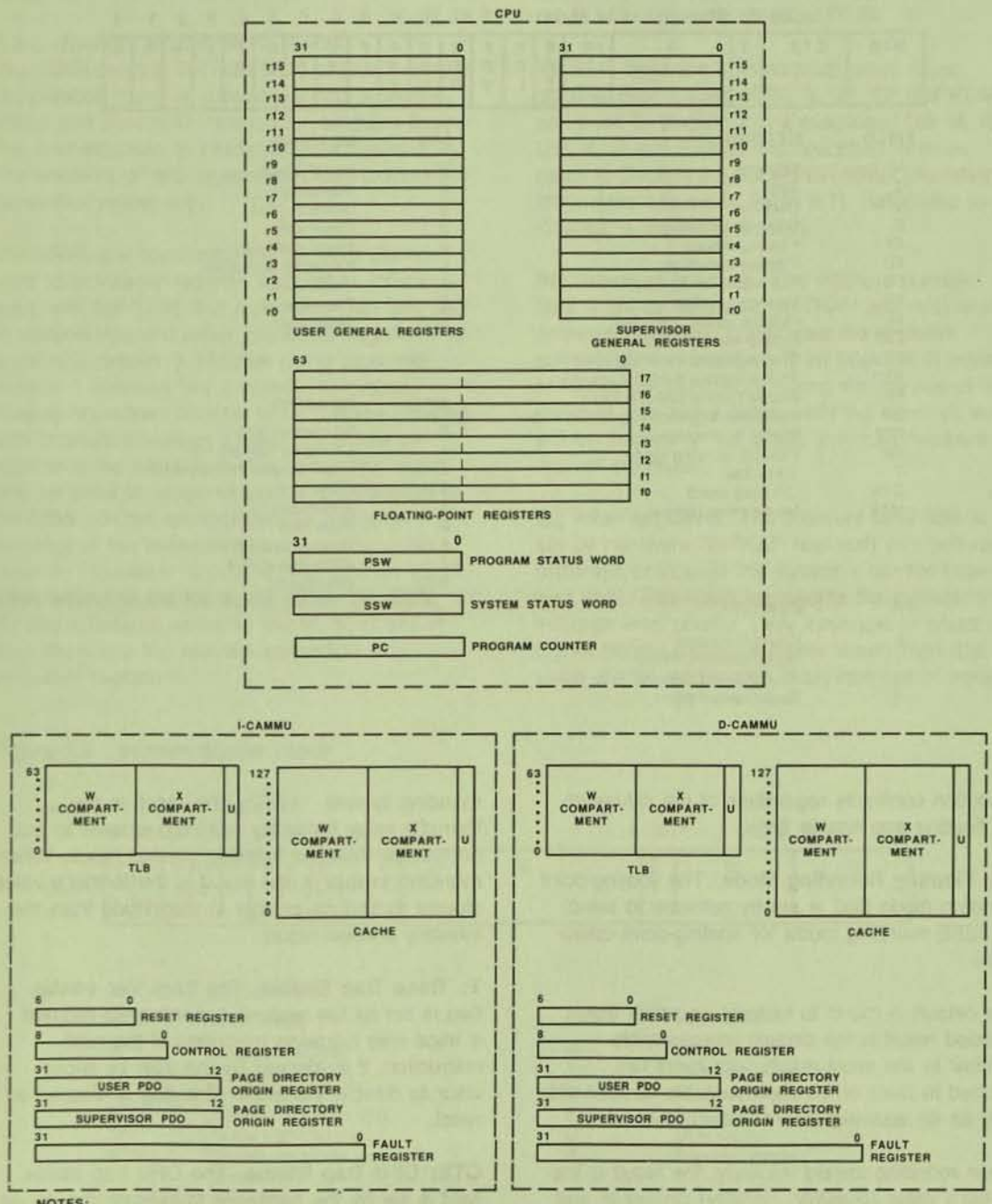
C,V,Z,N: Condition Codes. The condition codes are modified only by the register-to-register integer instructions, string comparison instructions, floating comparison and by directly writing the PSW. They are tested by the **branch on condition** instruction.

FX, FU, FD, FV, FI: Floating-Point Exception Flags. The floating-point exception flags are set by hardware on exceptions arising from floating-point operations, in accordance with the IEEE 754 Standard. Once set, they are cleared only by software or, for those conditions for which the corresponding trap is enabled (i.e., when both EFT and the individual enable flag are set) by the trap handler. They are tested by the **branch on floating exception** instruction.

EFX, EFU, EFD, EFV, EFI: Enable Floating Flags. The IEEE floating-point trap enable flags are set by software to request the result that would be given to a trap handler on an exception, rather than the IEEE default (no-trap) result. If the EFT flag is set, enabled exceptions also cause traps.

EFT: Enable Floating Trap. When set, the enable floating trap flag enables traps to occur whenever an exception condition is signalled by the FPU, and that condition's individual trap enable flag is also set. When this flag is clear, floating-point traps are disabled and program

Figure 2.1 CLIPPER Register Sets



NOTES:
CPU registers are addressed by instruction field.
I - CAMMU and D - CAMMU registers are located in supervisor virtual memory space.

Figure 2.2 Program Status Word

FIELD	MEANING
N	Negative
Z	Zero
V	Overflow
C	Carry out or borrow in
FX	Floating inexact
FU	Floating underflow
FD	Floating divide by zero
FV	Floating overflow
FI	Floating invalid operation
EFX	Enable floating inexact trap
EFU	Enable floating underflow trap
EFD	Enable floating divide by zero trap
EFV	Enable floating overflow trap
EFI	Enable floating invalid operation trap
EFT	Enable floating trap
FR	Floating rounding mode
T	Trace trap
CTS	CPU trap status
MTS	Memory trap status

CTS	MEANING
0	No CPU trap
1	(Reserved)
2	Divide by zero
3	(Reserved)
4	Illegal operation
5	Privileged instruction
6	(Reserved)
7	Trace trap
8-15	(Reserved)

MTS	MEANING
0	No memory trap
1	Corrected memory error
2	Uncorrectable memory error
3	(Reserved)
4	(Reserved)
5	Page fault
6	Read or execute protect fault
7	Write protect fault
8-15	(Reserved)

FR	MEANING
0	Round to nearest
1	Round toward + infinity
2	Round toward - infinity
3	Round toward zero

execution continues regardless of the values of the floating trap enable flags.

FR: Floating Rounding Mode. The floating-point rounding mode field is set by software to select the IEEE rounding mode for floating-point operations.

The default is round to nearest, in which the rounded result is the closest representable number to the exact result, with exact ties decided in favor of the representable number with zero as its least-significant fraction field bit.

When rounding toward +infinity, the result is the format's value (possibly +infinity) closest to and no less than the infinitely precise result. When rounding toward −infinity, the result is the format's value (possibly −infinity) closest to and no greater than the infinitely precise result. When rounding toward 0, the result is the format's value closest to and no greater in magnitude than the infinitely precise result.

T: Trace Trap Enable. The trace trap enable flag is set by the user or supervisor to request a trace trap following execution of the next instruction. It is cleared by the user or supervisor to disable the trace. This flag is cleared on reset.

CTS: CPU Trap Status. The CPU trap status field is set by the hardware to indicate the cause of a CPU-related trap.

MTS: Memory Trap Status. The memory trap status field is set by the hardware to indicate the cause of a memory-related trap.

2.2.4 System Status Word (SSW)

The SSW controls the CLIPPER Module's mode of operation (user or supervisor) and provides status and control information for program protection and response to interrupts (see Figure 2.3). The contents of this register can be modified in supervisor mode only.

The SSW is written using the **movwp** (move word to processor register) instruction. When used with the SSW, this instruction can take as its second operand either processor register 1 or processor register 3. **movwp** using processor register 1 behaves like a branch instruction, causing the upper pipeline to be flushed. **movwp** with processor register 3 does not cause the pipeline to be flushed, is thus faster, but must only be used in cases where the modification to the SSW will not compromise the memory mapping of the subsequent code in the upper pipeline. Processor register 3 may only be used when writing to the IN, IL, EI, FRD, TP, ECM, KU and UU flags; writes to the M, K, U and P flags must use the **movwp** instruction with processor register 1.

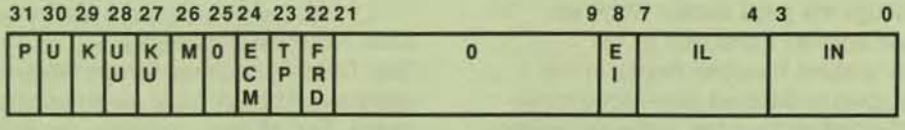
Reset clears the following SSW flags: EI, TP, M, U, K, KU, UU and P. The remaining bits are undefined. This represents unmapped supervisor mode with interrupts disabled.

All SSW flags are summarized below. More detailed descriptions of IN, IL, EI, TP and ECM are given in Section 2.6, Exceptions. The M, KU, UU, K, U and P flags are described in more detail in Section 3.5, CLIPPER MMU Implementation (see especially Table 3.1). Refer also to Chapter 9, Software Design.

IN: Interrupt Number. The interrupt number field is set by hardware (INTRAP and **reti**) and software (**movwp**). It indicates the system's current interrupt number. If an interrupt of equal or higher priority occurs during the service of an interrupt, this value (along with the interrupt level) will be pushed on the stack. Interrupt numbers are not prioritized.

IL: Interrupt Level. The interrupt level field is set by hardware (INTRAP and **reti**) and software (**movwp**) to indicate the system's current interrupt level. This value represents the current interrupt level priority. Only interrupts of equal or higher priority (equal or lesser value) than this value will be recognized. If an interrupt of equal

Figure 2.3 System Status Word



FIELD	MEANING
IN	Interrupt number
IL	Interrupt level
EI	Enable interrupts
FRD	Floating registers dirty
TP	Trace trap pending
ECM	Enable corrected memory error

FIELD	MEANING
M	Mapped mode
KU	User protect key
UU	User data mode
K	Protect key
U	User mode
P	Previous mode

or higher priority occurs during an interrupt service routine, this value will be pushed on the stack, and this field will be updated with the new interrupt level read from the Exception Vector Table by INTRAP.

EI: Enable Interrupts. The enable interrupt flag is set by hardware and software to enable interrupts and cleared by hardware and software to disable interrupts.

FRD: Floating Registers Dirty. The floating register dirty flag is set by hardware whenever a floating-point register is written. This flag is cleared by software. Its purpose is to permit operating systems to reduce context switching overhead by saving floating-point registers only if they have been written.

TP: Trace Trap Pending. The trace trap pending flag is automatically set by hardware whenever a trap or interrupt occurs during an instruction and the T flag is set. This ensures that the trace trap will be taken immediately after the trap or interrupt handler has finished executing, and that a single instruction is traced exactly once.

ECM: Enable Corrected Memory Error Trap. The enable corrected memory flag is set by software to enable a trap whenever a corrected memory error occurs.

M: Mapped Mode. The map flag is set by software to cause all address references to be mapped through the page tables. When set, virtual to real address translation by the CAMMUs is enabled (mapped mode). When cleared, mapping is disabled (unmapped mode) except for the first eight pages of the supervisor address space, which are always mapped via the Hardwired TLB.

U: User Mode. The user mode flag is set by the supervisor to indicate user mode of operation, and cleared to indicate supervisor mode of operation.

K: Protect Key. The protect key flag is set and cleared by the supervisor to select one of two sets of memory access protection codes. This flag is used for the access protection code selection in user mode, and in supervisor mode when the UU flag is clear.

KU: User Protect Key. The user protect key flag is set and cleared by the supervisor program to select one of two sets of memory access protection codes for memory access protection. This flag is used for the access protection code selection only during supervisor program execution when the UU flag is set.

UU: User Data Mode. The user data mode flag is set and cleared by the supervisor to select either supervisor or user data address space access when memory data is referenced in supervisor mode, and to select either the K or KU key flags for selection of the access protection codes used during supervisor memory references. When the UU flag is set, supervisor data references access user data space, and the KU flag is used for access protection code selection. When the UU flag is clear, supervisor data references access supervisor data space, and the K flag is used for access protection code selection. This flag is significant only in supervisor mode.

P: Previous Mode. The previous mode flag is copied from the U flag whenever the INTRAP sequence is executed. It is not overwritten when the new SSW flag is read from the Exception Vector Table, so as to indicate the operating mode of the program causing the exception.

2.2.5 CAMMU Registers

Each CAMMU contains five software-accessible registers, which are used for initialization and control. Two of these registers, the Supervisor PDO and User PDO, are used in address translation; they contain the base addresses of the supervisor and user page table directories (see Section 3.5, CLIPPER MMU Implementation). The Fault register is loaded with the virtual address associated with certain fault conditions and is used by the operating system to implement virtual memory. The Control and Reset registers are used to control CAMMU operation. The CAMMU registers are described in Section 3.5.2, CAMMU Register Descriptions.

2.3 DATA TYPES

The CLIPPER Module supports ten basic data types, as shown in Figure 2.4. Signed and unsigned bytes, halfwords (16 bits), words (32 bits), and longwords (64 bits) are all available, along with single-precision (32-bit) and double-precision (64-bit) IEEE Standard floating-point numbers.

The basic data types can be used to build complex structured data types, such as arrays and records. CLIPPER provides several addressing modes (see below) that facilitate accessing structured data types.

The rules for the alignment of data in memory are illustrated in Figure 2.5 and summarized below:

1. Bit 0 is the least-significant bit of all data types. Bit numbers increase from right to left.
2. The least-significant byte of multiple-byte data types is stored at the lowest memory address.
3. The most-significant byte of multiple-byte data types is stored at the highest memory address.
4. All data types must begin at an address that is a multiple of their size. For example, a halfword must begin on a halfword boundary.

2.4 INSTRUCTIONS

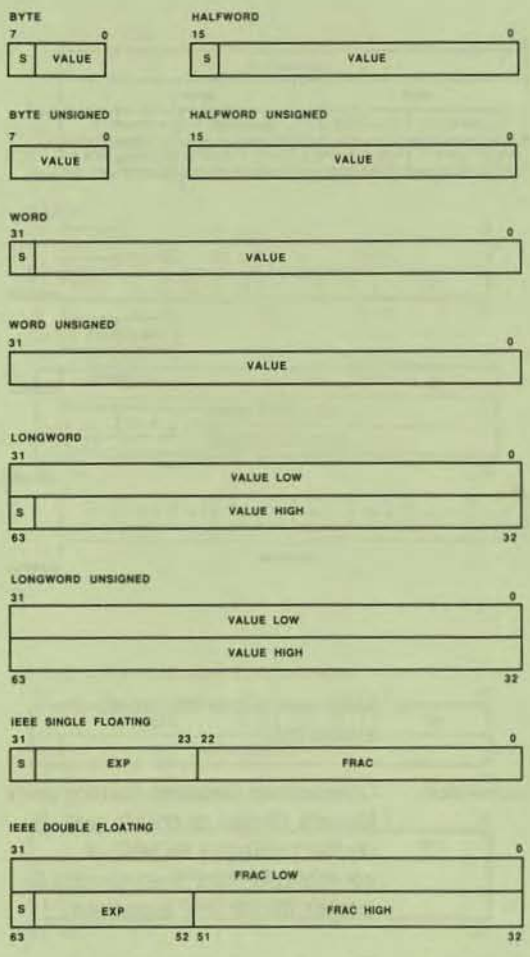
The CLIPPER instruction set contains 101 hardwired instructions and 67 macro instructions for operating on the basic data types. All CLIPPER instructions are described individually in Chapter 10, CLIPPER Instruction Set and summarized in Appendix B, CLIPPER Instruction Set Summary.

There are ten basic categories of instructions:

load/store — These instructions transfer addresses, bytes, halfwords, words, longwords, and floating point quantities between memory and registers.

move — These instructions transfer 32- and 64-bit quantities between registers.

Figure 2.4 CLIPPER Primitive Data Types

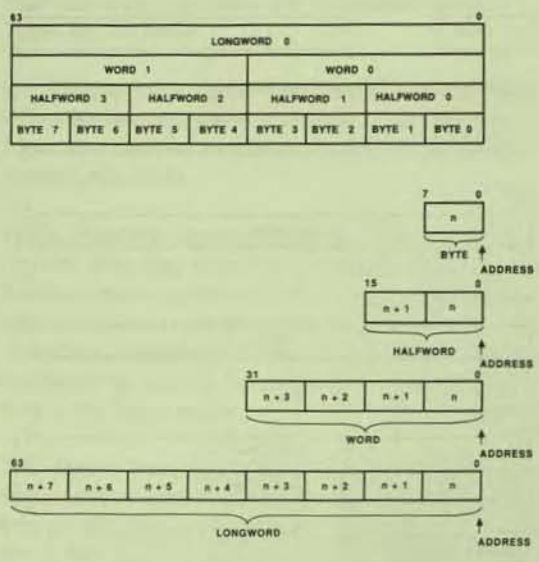


arithmetic — Add, subtract, multiply, divide, negate, modulus, and scale (multiply by a power of 2) are supported for registers or variable-length immediate values.

logical — And, or, xor, and not are supported for registers.

shift/rotate — Arithmetic and logical shifts (the difference lies in their treatment of right shifts for signed quantities) of

Figure 2.5 Alignment of Data in Memory



both words and longwords are supported.

conversion — Conversions between floating-point formats (single to double and double to single) as well as conversions from floating-point to integer format are supported.

compare — The values of words and both precisions of floating-point numbers can be compared; an atomic test-and-set instruction is also available.

string — These instructions manipulate character strings; compare, initialize, and move are provided.

stack — These instructions manipulate the stack; push and pop, along with save multiple registers and restore multiple registers are available.

control — Branches, call, call supervisor, returns, and noop are provided.

The 67 macro instructions are implemented in the Macro Instruction Unit as sequences of hardwired instructions. Except for their distinctive bit encoding, nothing distinguishes the macro instructions from hardwired instructions as far as the programmer is concerned. The macro instructions are scattered over the ten categories; for example, all conversions and string instructions are macros, along with most stack instructions (except **pushw**, **popw**, **call** and **ret**), and also one or two load/store, move, arithmetic, and control instructions. Six of the instructions (all macros) are *privileged*, that is, they can only be accessed by a program in supervisor mode (see Section 2.5).

Each instruction specifies an operation to be performed, and the type and location of the operands. These operands can be located either in memory, in a register, or in the instruction itself. To facilitate rapid instruction decoding, all instructions are built of halfword units called *parcels*. Depending on the instruction, from one to four parcels may be used.

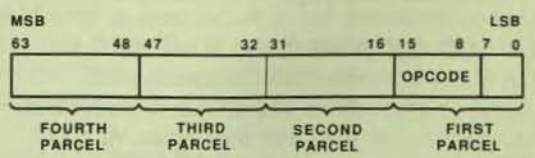
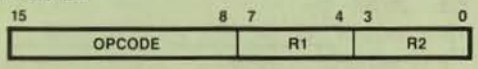


Figure 2.6 shows the formats of all CLIPPER instructions. As shown in the Figure, instruction formats fall into two groups: those with addresses and those without. The former are the instructions that need to access memory (primarily loads, stores, and branches); the latter are the arithmetic/logical instructions that generally execute in a single clock cycle. CLIPPER instructions have zero, one, or two operands, but only one operand can be accessed by a memory address.

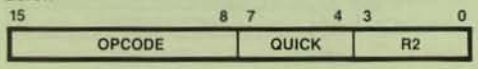
Figure 2.6 Instruction Formats

INSTRUCTION FORMATS — NO ADDRESS

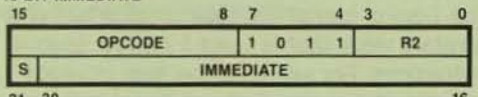
REGISTER



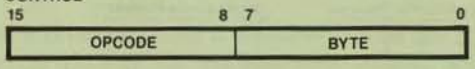
QUICK



16-BIT IMMEDIATE

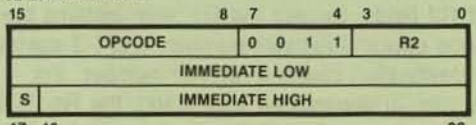


CONTROL



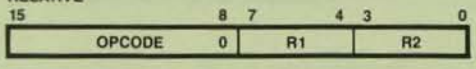
MACRO

32-BIT IMMEDIATE

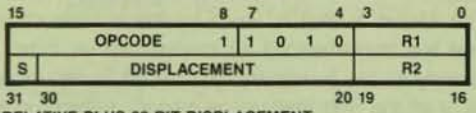


INSTRUCTION FORMATS — WITH ADDRESS

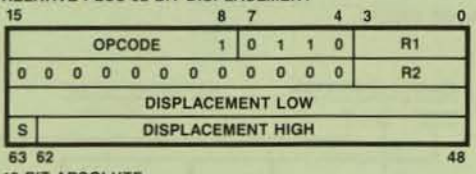
RELATIVE



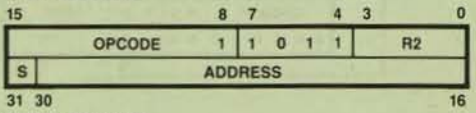
RELATIVE PLUS 12-BIT DISPLACMENT



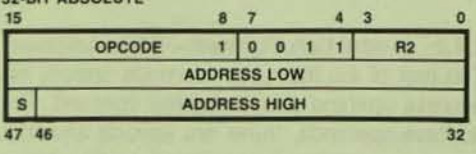
RELATIVE PLUS 32-BIT DISPLACEMENT



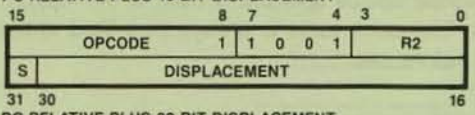
16-BIT ABSOLUTE



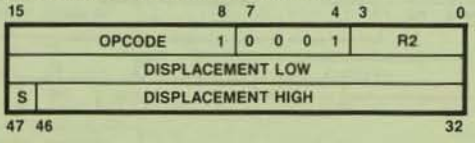
32-BIT ABSOLUTE



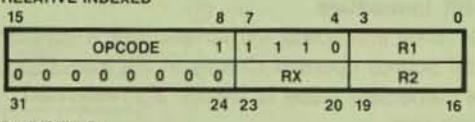
PC-RELATIVE PLUS 16-BIT DISPLACEMENT



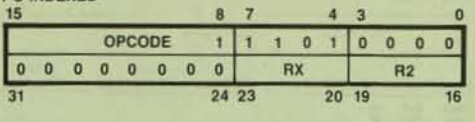
PC-RELATIVE PLUS 32-BIT DISPLACEMENT



RELATIVE INDEXED

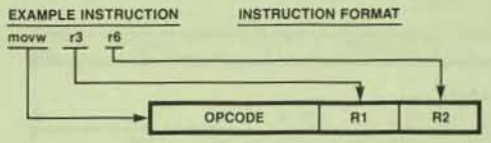


PC INDEXED



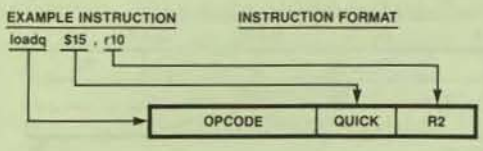
2.4.1 Instruction Formats—No Address

Register
The Register format is used for most instructions that take just one or two register arguments.

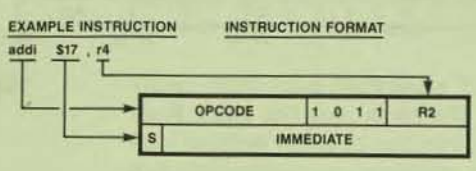


The opcode specifies the interpretation of the R1 and R2 fields. Usually the R1 field contains the source operand register number, and R2 contains the destination operand register number. For example, in the **movsw** instruction, the R1 field contains the number of the single-precision floating point register containing the source operand, and the R2 field contains the number of the general register in which to store the result.

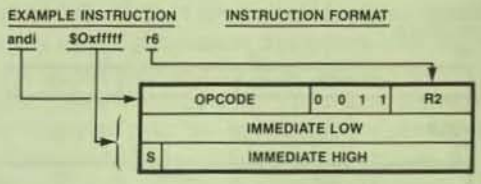
Quick
The Quick format encodes a constant, 4-bit unsigned source operand directly in the instruction. The quick value is always zero-filled at the left before use.



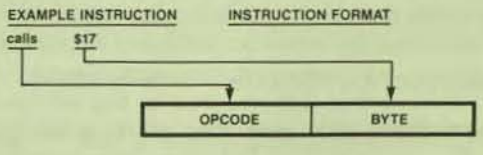
16-Bit Immediate
The 16-bit immediate format encodes a constant 16-bit source operand directly in the instruction. The immediate value is always sign-extended before use.



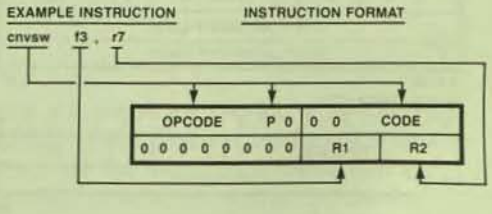
32-bit Immediate
The 32-bit immediate format encodes a constant 32-bit source operand directly in the instruction.



Control
The Control format encodes up to 8 bits of a constant value that is used by a few special instructions. For example, the byte operand specifies the system call number in the **calls** instruction.



Macro
The Macro format is used by those instructions that are implemented as macros rather than directly in the hardware. The P bit in the opcode, bit 9 of the first instruction parcel, selects a privileged macro.



2.4.2 Instruction Formats—With Address

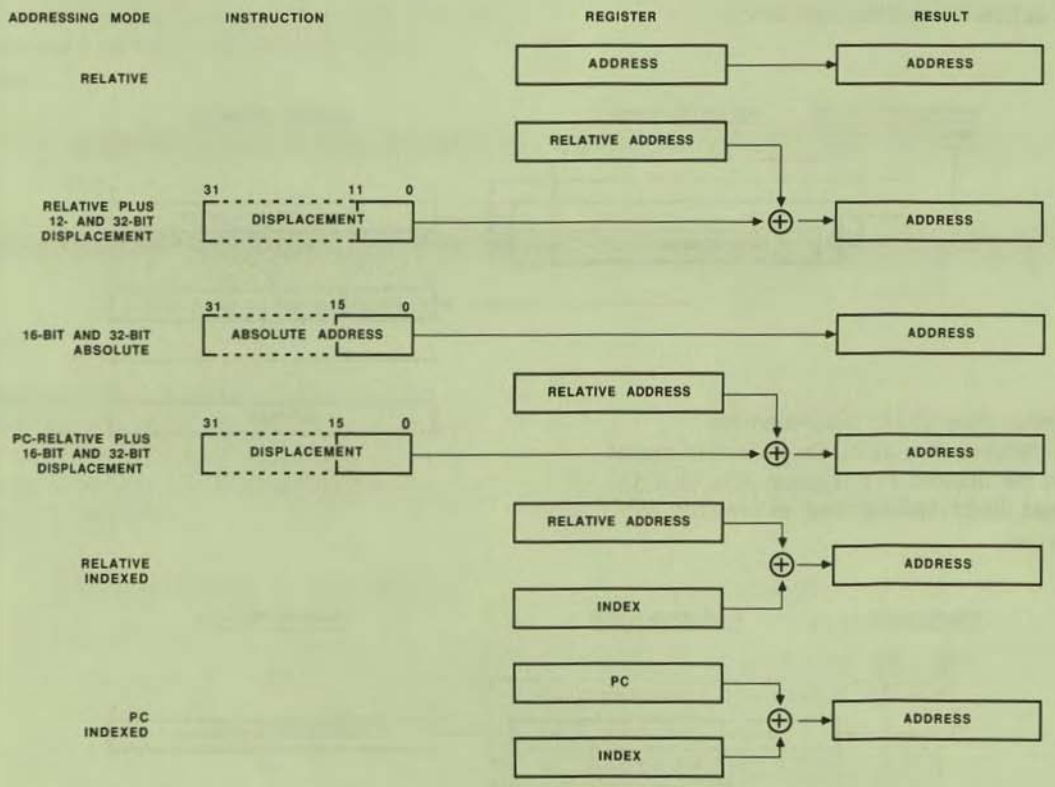
The rest of the instruction formats specify an address operand and a register operand. For address operands, there are several alternative ways of accessing them, called *addressing*

modes. An addressing mode is basically just a way of specifying an address as the sum of several factors, which may be stored in registers or provided with the instruction itself. The CLIPPER supports nine memory addressing modes, which are shown diagrammatically in Figure 2.7.

With the relative mode and the two relative with displacement modes, the address is either in a general register or is computed from the sum of the contents of a general register and a displacement value carried with the instruction. The two absolute modes carry the address as a pure displacement value with the instruction. The two PC relative modes are useful for branches relative to the current value of the PC. The two indexed modes compute the address by summing the contents of two registers.

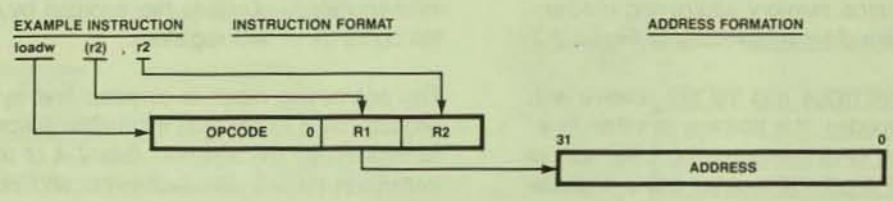
The addressing mode is selected first by the opcode (bit 8 of the first instruction parcel), and if necessary, by the AM field (bits 7:4 of the first instruction parcel). Displacements and absolute addresses are always sign extended.

Figure 2.7 Addressing Modes



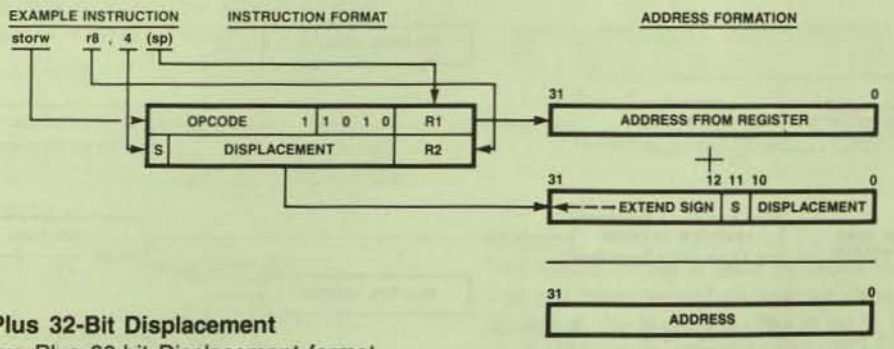
Relative

The Relative format uses the address in a Register (R) to compute an address.



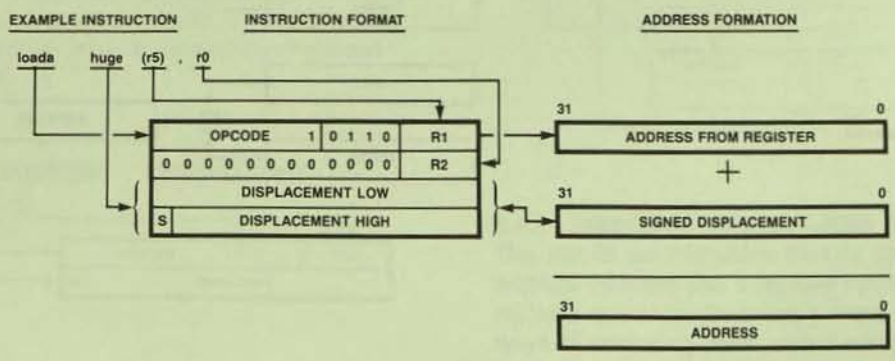
Relative Plus 12-bit Displacement

The Relative Plus 12-bit Displacement format uses the address in a register (R1), plus a signed 12-bit displacement, to compute an address. The displacement is sign-extended to 32 bits before the address calculation.



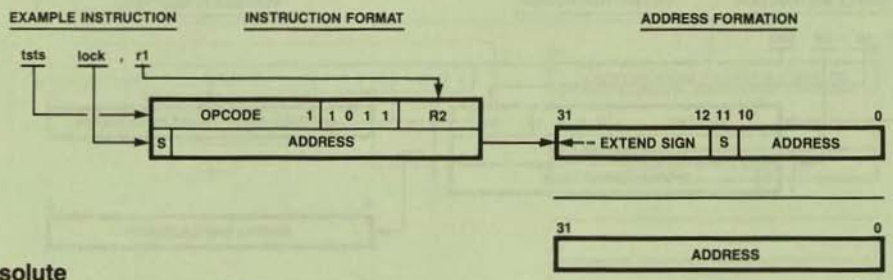
Relative Plus 32-Bit Displacement

The Relative Plus 32-bit Displacement format uses the address in a register (R1), plus a signed 32-bit displacement, to compute an address.



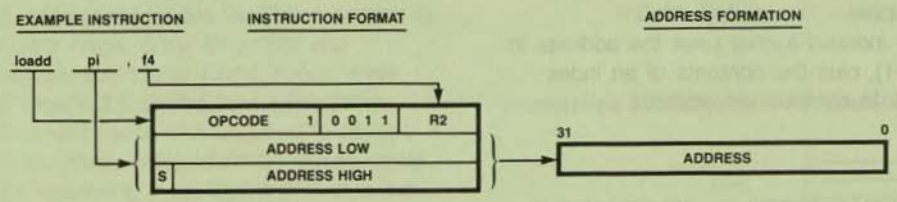
16-Bit Absolute

The 16-bit Absolute format uses the address in a register (R2).The address is sign-extended to 32 bits before being used. Because the address field is signed, the range of address that can be accessed with this format is 0xffff8000 ≤ address ≤ 0xffff.



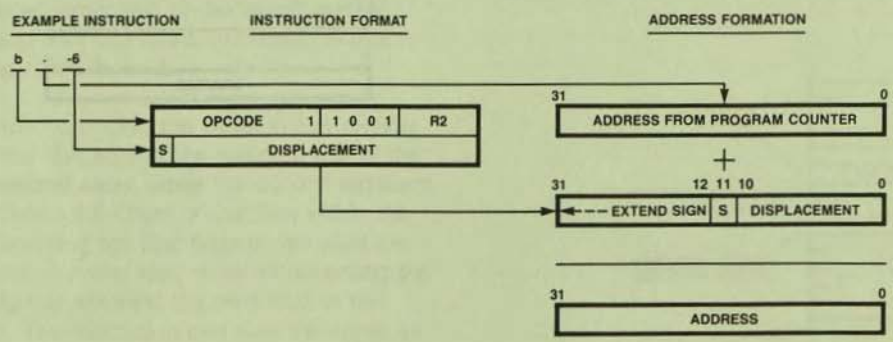
32-Bit Absolute

The 32-bit Absolute format uses the signed 32-bit displacement portion of the instruction as an address.



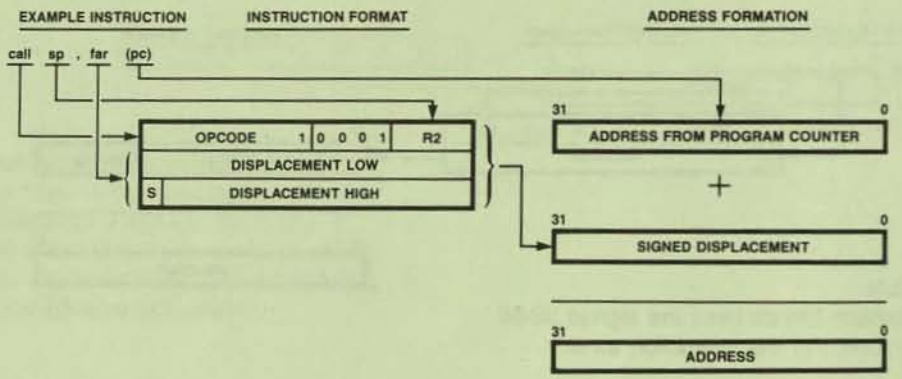
PC Relative Plus 16-Bit Displacement

The PC Relative Plus 16-bit Displacement format adds a signed 16-bit displacement to the contents of the Program Counter (PC) to compute an address.



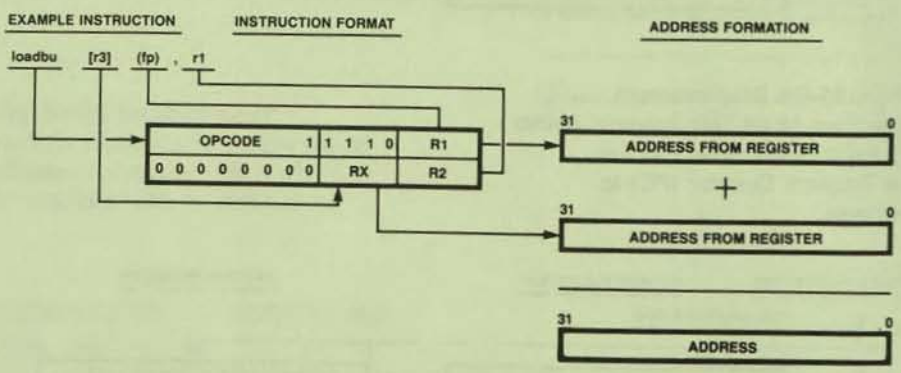
PC Relative Plus 32-Bit Displacement

The PC Relative Plus 32-bit Displacement format adds a signed 32-bit displacement to the contents of the Program Counter (PC) to compute the address.



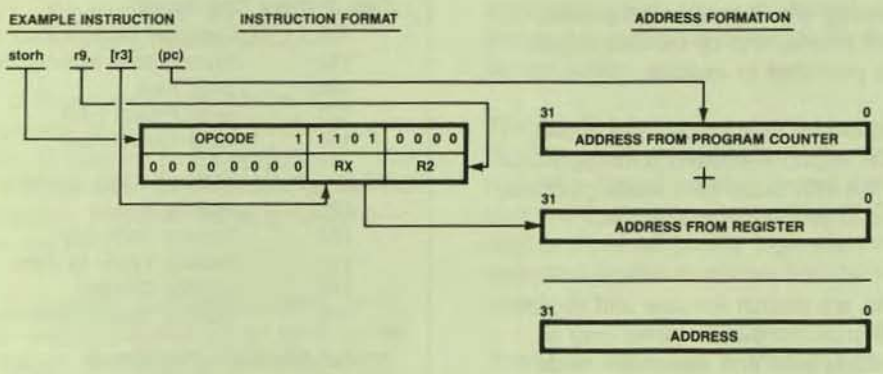
Relative Indexed

The Relative Indexed format uses the address in a register (R1), plus the contents of an index register (RX), to compute an address.



PC Indexed

The PC Indexed format adds the contents of an index register (RX) to the contents of the Program Counter (PC) to compute an address.



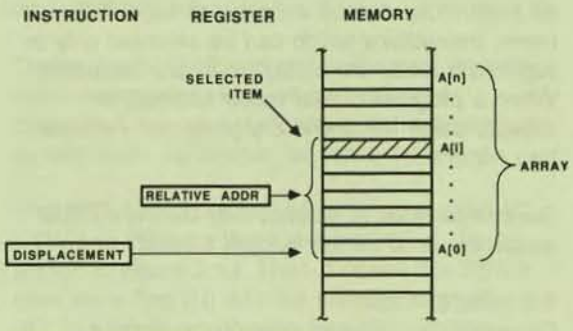
2.4.3 Addressing Mode Applications

Use of the addressing modes facilitates access to *structured data types*, such as arrays and records, that are commonly found in high-level languages. Figure 2.8 shows how the relative plus displacement mode can be used to access an entry in an array, while relative indexed mode is useful for accessing a particular item in a two-dimensional array.

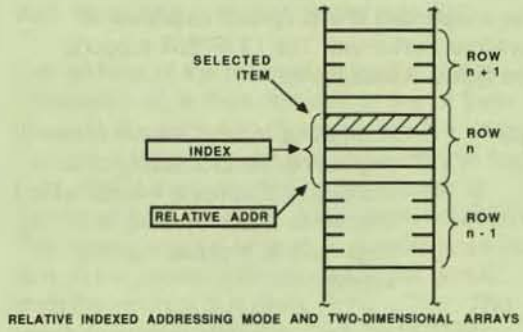
In the first example, the displacement value points to the base of the array, while the value in the register defines the offset of the item of interest. After the register is incremented by a fixed amount, the same addressing mode will point to the next item in the array; thus a sequence of items from an array can be accessed quickly using a loop with one basic instruction whose addressing mode is fixed.

In the second example, the value in one register points to the first item in the selected row of the two-dimensional array, while the second register's contents define the offset of the item within the row. Incrementing the first register will yield the same offset in a new row, while incrementing the second register will yield the next item in the same row. The instruction can stay the same as a whole set of two-dimensional arrays are accessed simply by incrementing registers.

Figure 2.8 Addressing Modes and Structured Data Types



RELATIVE ADDRESSING MODE AND ARRAYS



RELATIVE INDEXED ADDRESSING MODE AND TWO-DIMENSIONAL ARRAYS

2.5 SUPERVISOR AND USER MODES OF OPERATION

The CLIPPER Module has two modes of operation, *supervisor mode* and *user mode*, as selected by the SSW's U flag. The two modes are distinguished by the registers and address space they can access, and by the instructions which they are permitted to execute.

Programs executing in user mode have access only to the user registers, floating-point registers, the PSW and the PC. Supervisor mode programs have access to all programmer-accessible registers.

Address spaces are distinct for user and supervisor modes. Different translation tables may be used for translating user and supervisor mode addresses, and the access protection provided by the memory management mechanism allows access by supervisor programs to memory locations that are prohibited from access by user mode programs and vice versa.

All instructions can be executed in supervisor mode. Instructions which can be executed only in supervisor mode are called *privileged instructions*. When a program in user mode attempts to execute these instructions, a privileged instruction trap occurs.

Supervisor mode is entered only via the INTRAP sequence, or when the system is reset.

2.6 EXCEPTIONS

Exceptions are internal hardware conditions, external events, or even particular instructions, that cause the normal operation of the processor to be suspended and a special sequence of operations performed. The CLIPPER supports three types of exceptions:

traps — Anomalous internal events occurring during the processing of an instruction. Examples include an attempt to divide by zero, or a page fault in a virtual memory system.

interrupts — An external device's method of signalling the CPU that it needs

Figure 2.9 Exception Vector Table

Real Address (Hex)	Description
Data Memory Trap Group:	
108	Corrected Memory Error
110	Uncorrectable Memory Error
128	Page Fault
130	Read Protect Fault
138	Write Protect Fault
Floating-Point Arithmetic Trap Group:	
180	Floating Inexact
188	Floating Underflow
190	Floating Divide by Zero
1A0	Floating Overflow
1C0	Floating Invalid Operation
Integer Arithmetic Trap Group:	
208	Integer Divided by Zero
Instruction Memory Trap Group:	
288	Corrected Memory Error
290	Uncorrectable Memory Error
2A8	Page Fault
2B0	Execute Protect Fault
Illegal Operation Trap Group:	
300	Illegal Operation
308	Privileged Instruction
Diagnostic Trap Group:	
380	Trace Trap
Supervisor Calls:	
400	Supervisor Call 0
408	Supervisor Call 1
.	
.	
.	
7F8	Supervisor Call 127
Prioritized Interrupts:	
800	Non-Maskable Interrupt
808	Interrupt Level 0 Number 1
810	Interrupt Level 0 Number 2
.	
.	
.	
878	Interrupt Level 0 Number 15
880	Interrupt Level 1 Number 0
888	Interrupt Level 1 Number 1
.	
.	
.	
FF8	Interrupt Level 15 Number 15

	servicing. For example, a DMA controller signalling that it has transferred a block of data into memory.
supervisor calls	Program generated requests for operating system services.

If any one of these exceptions occurs, it is usually necessary to immediately invoke a software handler to respond to the exceptional condition. It is this need for immediate action that makes exceptions suspend normal processing—they simply can't wait.

Pipelining complicates exception handling, since the pipeline must be cleared out as soon as the exception occurs, in order that the exception handler can be executed immediately after the instruction that was in the execution phase when the exception occurred. Then when the normal processing resumes, the program counter must be backed up to refetch the instruction immediately following the one that was executing when the exception occurred. Multiple exceptions present additional complications; for example, a floating-point trap at the same time as a page fault.

The CLIPPER architecture supports 18 traps, 256 vectored interrupts, and 128 programmable supervisor calls. The traps are caused by page faults, memory protection violations, floating-point exceptions such as overflow, integer arithmetic errors such as division by zero, and privileged instruction violation by a user mode program. When one or more of these conditions occurs, the hardware automatically generates the appropriate trap. Interrupts are signaled by activity on the interrupt pins, with the type of interrupt encoded as an eight bit quantity on the interrupt bus. Supervisor calls are made by executing the **calls** instruction with a parameter specifying which call.

No matter what their cause or type, all exceptions are handled in much the same way. The *Exception Vector Table*, located in the first real page of memory, contains the address and SSW value for every exception handler routine. These address/SSW pairs are stored in predefined locations, with each location corresponding to a particular type of trap, interrupt, or call. (Unassigned Exception Vector Table addresses are reserved for future use by Fairchild and must be initialized to point to a valid handler routine.) Figure 2.9 shows the layout of the Exception Vector Table.

The priority of exceptions is the order shown in the Exception Vector Table (in the order from highest to lowest priority), except that the trace trap has the lowest priority. The CLIPPER Module's internal priority logic ensures that exception service is always granted to the highest priority event.

There are two macro instruction sequences, INTRAP and **reti**, that manage the entry to and exit from all exceptions. The INTRAP sequence performs a context switch, and then transfers control to the trap or interrupt handler. The **reti** instruction is an interrupt/trap return, which restores the system to the correct user or supervisor environment.

During the INTRAP and **reti** sequences, as with most macro sequences, all interrupts are disabled; traps are not disabled, but only serious system faults can occur, as explained below.

The INTRAP sequence begins by saving the PC, SSW and PSW on the supervisor stack, as shown in Figure 2.10. Then it copies the SSW's user mode flag (U) into the previous mode flag (P). In order to access the Exception Vector Table, INTRAP sets the user mode flag to supervisor mode and clears the protect key (K), user data mode (UU) and user protect key (KU).

The address of the required Exception Vector Table entry, V, is then obtained in one of three ways: 1) For traps and the non-maskable interrupt, the address is generated from internal trap logic. 2) For supervisor calls, the address is generated from the lower 7 bits of the instruction. This value is multiplied by 8 and 400H is added to it. 3) For priority interrupts, a number is read from the Interrupt Bus Lines, $\overline{\text{IVEC}}$ <7:0>. This value is inverted, multiplied by 8, and 800H is

Figure 2.10 Supervisor Stack After INTRAP

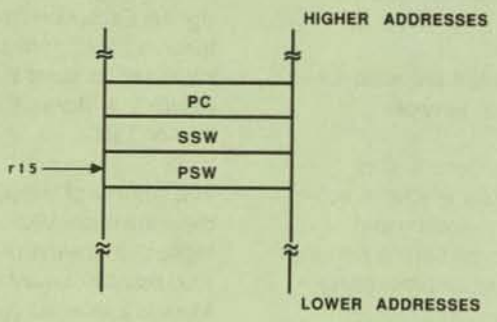


Figure 2.11 Calculation of Exception Vector Table Entry for Maskable Interrupts and Supervisor Calls

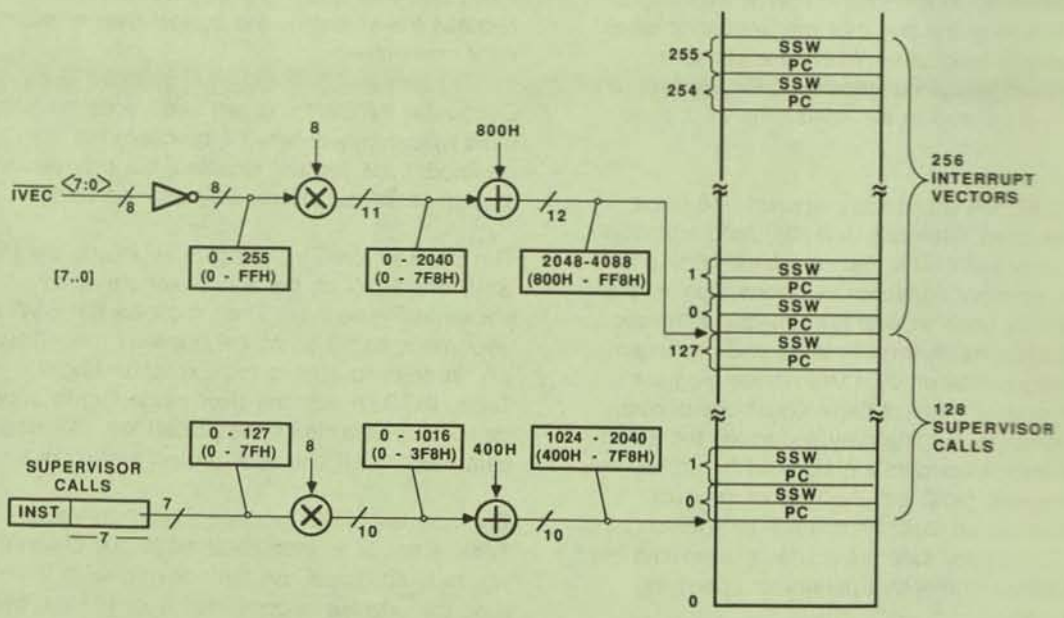


TABLE 2.1 Conditional and Unconditional Traps

Conditional Traps	Unconditional Traps
Corrected Memory Error	Uncorrectable Memory Error
Floating-Point Arithmetic Trap Group	Page Fault
Trace	Protect Faults
	Privileged Instruction
	Illegal Operation
	Integer Divide by Zero
	Supervisor Call

added to it. The calculation of the Exception Vector Table entry is shown in Figure 2.11.

INTRAP uses V to obtain the new PC value and V+4 to obtain the associated SSW value. The new SSW value is transferred to the SSW, overwriting the previous contents of SSW except for the previous mode flag (P), which is retained in order to indicate the mode of the interrupted program. INTRAP then exits, and control is passed to the trap or interrupt service routine. The PSW saved on the stack will have MTS, CTS or the floating-point exception flags set to indicate the cause of the trap.

After completing its service, the trap or interrupt handler executes the **reti** instruction, which restores PSW, SSW and PC to their contents prior to INTRAP.

The occurrence of a trap during INTRAP or **reti** results in an Unrecoverable Fault (URF). The CLIPPER Module halts in a controlled suspended state, drives the $\overline{\text{URF}}$ signal low as an alarm, and remains in that state until re-started by the $\overline{\text{RESET}}$ signal. (When $\overline{\text{URF}}$ is asserted, all inputs other than $\overline{\text{RESET}}$ are ignored.)

To avoid the occurrence of a page fault during INTRAP and **reti** (and the resulting $\overline{\text{URF}}$ condition), the supervisor stack must always have a valid page table entry that permits both reading and writing. This will prevent page faults from occuring during INTRAP or **reti**, because the supervisor stack is the only memory area referenced by these sequences.

2.6.1 Traps

Traps are exceptions recognized by the CPU during execution of single instructions (e.g., overflow, page fault). They can be signalled in the CPU or by either of the CAMMUs. There are 18 predefined traps, as shown in Figure 2.9.

A trap causes all instructions in the pipeline to either be backed out or completed in a consistent manner, depending on their position in the pipeline (refer to Chapter 9, Software Design, for a more detailed description of the pipeline's response to trap conditions).

Both conditional and unconditional traps are supported (see Table 2.1). Conditional traps are enabled by bit settings in the PSW and occur only when enabled.

Traps may be generated at various stages of instruction processing, as shown in Table 2.2. The CLIPPER Module's internal trap logic ensures that the saved program counter points to the instruction at which the trapped program may be correctly restarted.

2.6.1.1 Data Memory Trap Group

Data memory traps occur when the data cache interface reports a fault. These traps cause the faulting instruction to be aborted and subsequent

TABLE 2.2 Trap Handler Environment Summary

Trap	When Trap Is Taken	Return Address (Saved In Supervisor Stack)
Data Memory Trap Group	During Execution	Faulting Instruction
Floating-Point Arithmetic Trap Group	After Execution	Next Instruction To Be Executed
Integer Arithmetic Trap Group	After Execution	Next Instruction To Be Executed
Instruction Memory Trap Group	Before Execution	Faulting Instruction
Illegal Operation Trap Group	Before Execution	Faulting Instruction
Diagnostic Trap Group	After Execution	Following Instruction
Supervisor Call	After Execution	Following Instruction

instructions already in the upper pipeline to be backed out.

Data memory traps are recorded in the PSW's memory trap status (MTS) field. The MTS field is also used by the instruction memory trap group for the same fault conditions. Interpretation is unambiguous because instruction memory traps are deferred until data memory traps have been serviced, and they are serviced by different trap handlers.

With **pushw** and **popw** instructions, the stack pointer is decremented or incremented in parallel with the instruction's memory access. Thus, when a data memory trap occurs during a **pushw** or **popw** instruction, the operating system must, before restarting the program, restore the stack pointer to the value it had prior to the trapping instruction, i.e., decrement the stack pointer by 4 for **popw**, or increment the stack pointer by 4 for **pushw**.

Corrected/Uncorrectable Memory Errors. Corrected and uncorrectable memory errors are detected by memory and communicated to the cache via the two system bus signals, $\overline{\text{MSBE}}$/$\overline{\text{RETRY}}$ (corrected memory single bit error/ retry) and $\overline{\text{MMBE}}$ (uncorrectable memory multiple bit error). It is the responsibility of the memory subsystem to save the real memory address of the location that failed in a predetermined location, where it may be accessed for maintenance by the operating system.

The operating system may ignore indications of corrected memory errors by clearing the ECM flag in the SSW.

Page Fault. A page fault occurs when a program attempts to access a page for which there is no valid entry in the currently assigned page directory or page tables, as explained in Chapter 3. The operating system uses this fault to allocate pages to user or supervisor programs. The address saved on the supervisor stack is the program address of the instruction that caused the page fault. The virtual address of the data memory location that caused the fault is saved in the CAMMU's Fault register.

Read/Write Protection Faults. Read and write accesses to each page are validated by a comparison of the U, K, UU and KU flags in the SSW with the protection code in the TLB or page table entry. When an access violation occurs, the address saved on the supervisor stack is the program address of the instruction that caused the fault. The virtual address of the data memory location that caused the fault is saved in the CAMMU's Fault register.

2.6.1.2 Floating-Point Arithmetic Trap Group

There are five distinct floating-point exceptions which are specified in the IEEE Standard 754. These exceptions are signalled by the FPU in the case of invalid operation, inexact result, overflow result, underflow result, and divide by zero.

For each exception, there corresponds a *floating-point exception* flag in the PSW, which is set on any occurrence of the exception, and a *floating-point trap enable* flag, which determines whether the trap is taken and, if it is not taken, the returned result. There is also is a *floating-point group trap enable* flag. When an exception arises for which the individual trap enable flag is true (set to "1") and the group trap enable flag is also true, then a floating-point trap is invoked and control is transferred to a user-specified trap handler. If the group trap enable is false, or if the individual trap enable flag is false, then the trap is not invoked.

For the underflow and overflow exceptions, the behavior of the FPU is determined by the values of the floating-point trap enable flags as specified in the Standard. In particular, overflows with the overflow trap disabled deliver infinity or max_real, whereas the result with the trap enabled is the normalized result with the exponent distorted. Underflows are handled similarly.

The software knows which floating-point trap has occurred because each floating-point trap invokes a separate trap handler (i.e., each trap has its own entry in the Vector Table). It is not sufficient to examine the floating-point exception flags, since the state of these immediately before executing the exceptional operation is generally unknown. When the floating-point trap handler is

entered, the **loadfs (load floating status)** instruction is executed first to acquire useful information about the instruction causing the exception.

The floating-point exceptions are described in more detail in Section 6.9.2, Floating-Point Exceptions.

2.6.1.3 Integer Arithmetic Trap Group
The CPU trap status field in the saved PSW indicates the cause of the integer arithmetic trap.

Integer Divide by Zero. The integer divide by zero exception is signalled when an integer divide or modulus instruction is executed with zero divisor.

Integer divide by zero cannot be disabled. The result of the trapped instruction will not be written to the specified register.

2.6.1.4 Instruction Memory Trap Group;
Instruction memory faults are detected and signalled by the instruction interface. These traps are not acted upon when first sensed, i.e., if a branch instruction or other sequence altering event occurs between the time that the instruction interface detects the trap condition and when that instruction arrives at the issue phase of the pipeline, then the pending trap condition is cleared and the trap is deferred. A deferred trap will not trap until it is ready to be issued for execution. If preempted by another trap, it may trap later if the code is restarted.

The faulting instruction has not entered the lower pipeline when the trap is taken, and the program address saved is that of the faulting instruction.

For instruction memory traps, the memory trap status (MTS) field in the saved PSW indicates the reported error. Although the MTS field is also used for data memory traps, interpretation is unambiguous because instruction memory traps are deferred until data memory traps have been serviced, and they are serviced by different trap handlers.

Corrected/Uncorrectable Memory Error. Uncorrectable and corrected memory errors are detected by memory and communicated to the cache via the $\overline{\text{MSBE}}/\overline{\text{RETRY}}$ or $\overline{\text{MMBE}}$ bus signals. It is the responsibility of the memory system to save the real memory address of the location that failed in a predetermined location in memory, where it can be accessed for maintenance by the operating system.

The operating system may ignore indications of corrected memory errors by clearing the ECM flag in the SSW.

Page Fault. A page fault occurs when a program tries to access a page for which there is no valid entry in the currently assigned page directory or page tables. The operating system uses this fault to allocate pages to user and supervisor programs. The address saved on the supervisor stack is the program address of the instruction that caused the page fault. The virtual address of the memory location that caused the fault is saved in the CAMMU's Fault register.

Execute Protect Fault. Instruction fetches from each page are validated by a comparison of the U, K, UU and KU flags in the SSW with the protection code in the TLB or user's page tables. When an instruction fetch violation occurs, the address saved on the supervisor stack is the program address of the instruction that caused the fault. The virtual address of the memory location that caused the protection fault is saved in the CAMMU's Fault register.

2.6.1.5 Illegal Operation Trap Group
Illegal operation traps are taken before the instruction is executed. The program address saved on the supervisor stack is the address of the instruction which caused the trap. The CPU trap status field in the saved PSW indicates the nature of the trap.

Illegal Operation Fault. An illegal operation trap results from the attempted execution of any undefined instruction opcode or the occurrence of an addressing mode which is not specifically allowed.

Privileged Instruction Fault. A privileged instruction trap occurs when a privileged macro

instruction is encountered in user mode.

2.6.1.6 Diagnostic Trap Group

Trace Trap. Unless preempted by a another trap or interrupt, the trace trap occurs following the execution of an instruction whenever the PSW's T flag is set. For traced instructions which are interrupted or cause traps, the TP flag is set by hardware when the interrupt or trap occurs to ensure that the trace trap will occur immediately after the interrupt or other trap has been serviced.

Macro sequences are treated as a single instruction for trace purposes, so that the entire sequence executes before the trace trap is taken.

At the time of the trap, the CPU trap status field in the saved PSW indicates that a trace trap has occurred. The saved PC is the address of the instruction following the instruction that caused the trace trap.

2.6.1.7 Supervisor Call Traps

A supervisor call is an instruction executed as a trap, and is made using the **calls** instruction. Its purpose is to provide controlled access to system-level functions. There are 128 supervisor call codes, with separate Vector Table entries for each. The PC value saved on the stack is the address of the instruction following the **calls** instruction.

2.6.1.8 Multiple Traps

Only traps in the data memory and floating-point groups can be signalled at the same time. CLIPPER internal trap logic permits correct recovery of both faulting instructions. INTRAP transfers control to the floating-point trap handler, and the **loadfs** instruction is used to access the floating-point instruction that caused the trap. The MTS field in the saved PSW may be used by the floating-point trap handler to determine which data memory trap occurred.

2.6.2 Interrupts

Interrupts are events signalled by devices external to the CLIPPER Module and input to the module via the interrupt pins. Interrupts are taken when the following conditions are met:

- No traps are pending (except the trace trap).
- Interrupts are enabled.
- All instructions in the lower pipeline (except the string instructions) have finished executing.
- String instructions have either completed execution or have detected the interrupt and saved sufficient state information for continuation.

The CLIPPER supports 256 vectored interrupts. There are 255 maskable interrupts organized into 16 priority *levels*, and a single non-maskable interrupt. Interrupt levels are numbered in the order of decreasing priority. Thus level 0 has the highest priority and level 15 has the lowest priority. Each level contains interrupt *numbers* of equal priority. Level 0 contains 15 numbers; Levels 1-15 each contain 16 numbers. The non-maskable interrupt has a higher priority than all interrupt levels and, as its name implies, cannot be masked by software.

The CPU contains the logic necessary to arbitrate interrupt requests according to the priority of the interrupt level. The interrupt level currently being processed is stored in the interrupt level IL field of the SSW. The CPU accepts interrupts only for interrupt levels of equal or higher priority than the current interrupt level.

2.6.2.1 Maskable Interrupt Request/ Acknowledge Protocol

Interrupts are requested by the simultaneous activation of the $\overline{IRQ}$ input line and the assertion of the interrupt's *vector number* on $\overline{IVEC}$<7:0>. The vector number is composed of the interrupt level on $\overline{IVEC}$<7:4> and the interrupt number on $\overline{IVEC}$<3:0>.

An interrupt request will be acknowledged by the CPU if interrupts are enabled (the interrupt enable flag in SSW is set) and the interrupt level ($\overline{IVEC}$<7:4>) is of equal or higher priority than the interrupt level contained in the SSW's interrupt level (IL) field.

If the priority condition is met, a flag is set which causes the CPU to enter the pre-trap state, in which the execution pipeline is emptied by withholding issue of the instruction in the issue and control phase. The instructions in the execution

pipeline complete executing. If their execution causes a trap to be signalled, the interrupt is deferred until after the higher priority trap has been serviced.

More information on the protocol and timing for interrupts is provided in Section 7.2.7, Interrupt Bus and Control. Chapter 9 describes the software considerations for interrupt service routines.

2.6.2.2 Non-Maskable Interrupts

The non-maskable interrupt is signalled by the $\overline{NMI}$ input to the CPU. If $\overline{NMI}$ is asserted after another interrupt has already been acknowledged, the non-maskable interrupt request remains pending and is serviced following the INTRAP sequence for the acknowledged interrupt.

Chapter 3
Memory Management

3.1 INTRODUCTION

The real main memory of most computers is organized as a set of consecutively numbered storage cells, each containing a byte of data. The location numbers associated with these storage cells are called *real addresses*, and the set of all the real addresses is called the *real address space* of the computer. The real address space is thus determined by the actual hardware of the computer's memory system.

On the other hand, a program running on the computer generates a set of addresses as it executes, and this set is limited only by the maximum number of bits possible in an address. This set of possible addresses is called the *virtual address space* of the computer. Note that the virtual address space need not be the same size as the real address space; in fact the virtual space is usually much larger. For example, consider a CLIPPER system configured with 4 million bytes of memory. A program on this computer can address more than 4 billion locations, for the simple reason that 4,294,967,296 (2^{32}) 32-bit numbers exist. Thus the virtual address space is a thousand times larger than the real memory.

Memory management is concerned with the organization of a computer's virtual and real address spaces, how they are related to each other, and how memory can most effectively be used. This chapter will discuss three aspects of the CLIPPER's memory management scheme:

Memory Architecture: The organization of the virtual and real address spaces.

Address Translation: The mapping between virtual and real addresses.

Virtual Memory: Architectural mechanisms in the CLIPPER that permit use of the complete virtual address space in systems with limited real memory.

3.2 MEMORY ARCHITECTURE

The CLIPPER memory architecture is defined by the layouts of real address space and virtual address space. Figure 3.1 shows a diagram of the CLIPPER memory architecture.

3.2.1 Real Address Space

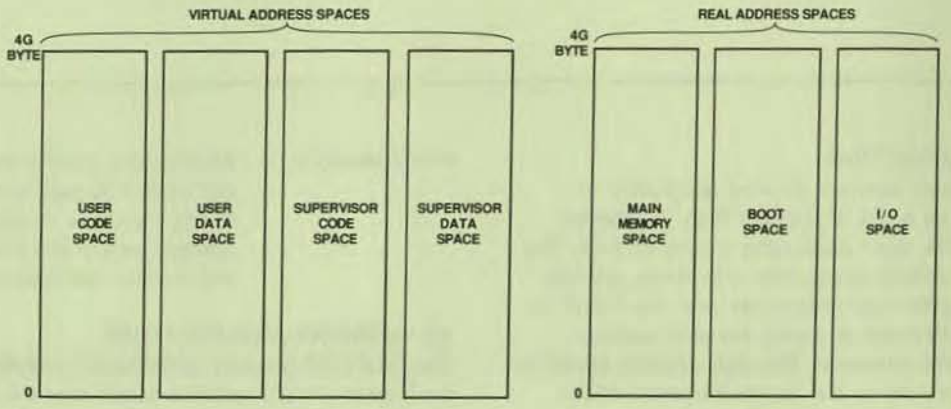
The maximum amount of real memory that can usefully be connected to a computer is determined by the number of address lines in the system bus. For the CLIPPER this number is 32, so the CLIPPERs maximum real memory and thus its maximum real address space is 4 gigabytes (2^{32} bytes).

The CLIPPER puts additional information on the system bus along with the real address; one important item is a 3-bit quantity called the *system tag*. The tag will be described in more detail below, but one of its intended purposes is relevant here, namely, to distinguish three separate real address spaces:

- main memory space
- boot space
- I/O space

Main memory space is the usual read/write memory. Boot space is a separate memory containing initialization code that is automatically activated when the computer system is reset. Typically this memory space will be implemented as a ROM. It is useful to have these two be separate address spaces, so that the boot code can start at real location 0 (supervisor virtual address 6000 hex) without either permanently taking

Figure 3.1 CLIPPER Memory Architecture



up a chunk of the lowest addresses in main memory or requiring sophisticated circuitry to switch it in and out of main memory as required. CLIPPER provides this switching logic on chip, so the memory control logic need only decode the system tag and send addresses to main memory or to boot space as appropriate.

The CLIPPER uses *memory-mapped I/O*, that is, it does not employ special I/O instructions. Instead, the usual loads and stores that are employed to access memory are also used for input and output. The I/O devices on the system bus are responsible for recognizing which addresses are intended for them. In a CLIPPER system, I/O has its own 4 gigabyte address space, which doesn't interfere with the 4 gigabyte main memory space. The I/O devices and memory control logic only have to decode the system tag to determine which of them is the intended recipient of an address.

3.2.2 Virtual Address Space

The virtual address space of the CLIPPER is a linear, unsegmented 4 gigabyte (2^{32} bytes) space. Just as the system tag on the bus effectively creates three separate real address spaces, so the internal operating modes of the CLIPPER create different virtual address spaces. It is useful to think of the CLIPPER as having four virtual address spaces at any one time: *user instruction space*, *user data space*, *supervisor instruction space*, and *supervisor data space*. Programs running in user mode generate addresses in one of the two user spaces (just as they access the user registers), while supervisor mode programs access the two supervisor spaces. As described in the next section, the CLIPPER provides hardware mechanisms that support these four separate spaces.

3.3 ADDRESS TRANSLATION

Address translation is the process which maps virtual address space onto real address space. *Mapping* is the address translation scheme that allows virtual addresses to be translated into arbitrary real addresses; it provides a kind of generalized relocation mechanism.

Unmapped memory systems simply equate virtual addresses to real addresses—virtual address 00001234 becomes real address 00001234. This approach is appropriate for simple, single-user or embedded systems. Unmapped mode can be selected on the CLIPPER by setting a flag in the SSW, though on the CLIPPER, the first eight supervisor pages are permanently mapped via the HTLB, as explained in Section 3.5.1. The basic mapping of addresses in mapped and unmapped modes is shown in Figure 3.2.

Figure 3.2 Main Memory Translations—Mapped/Unmapped Modes

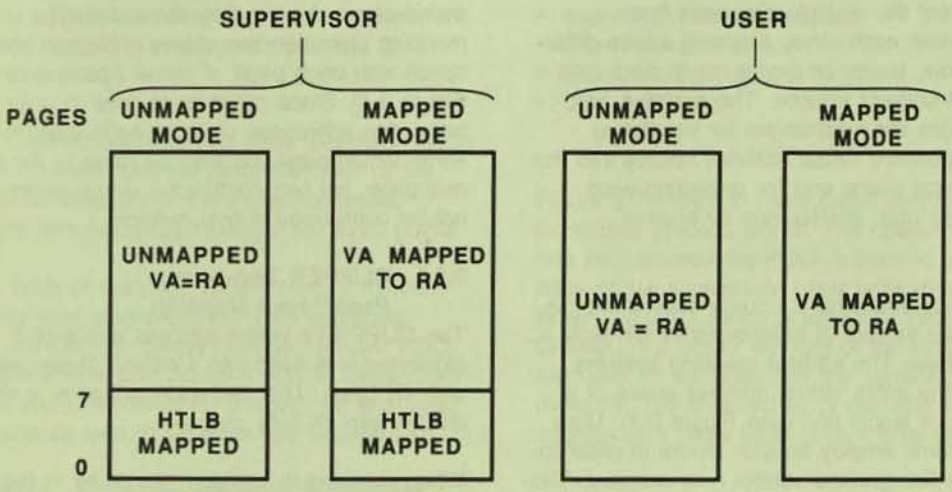
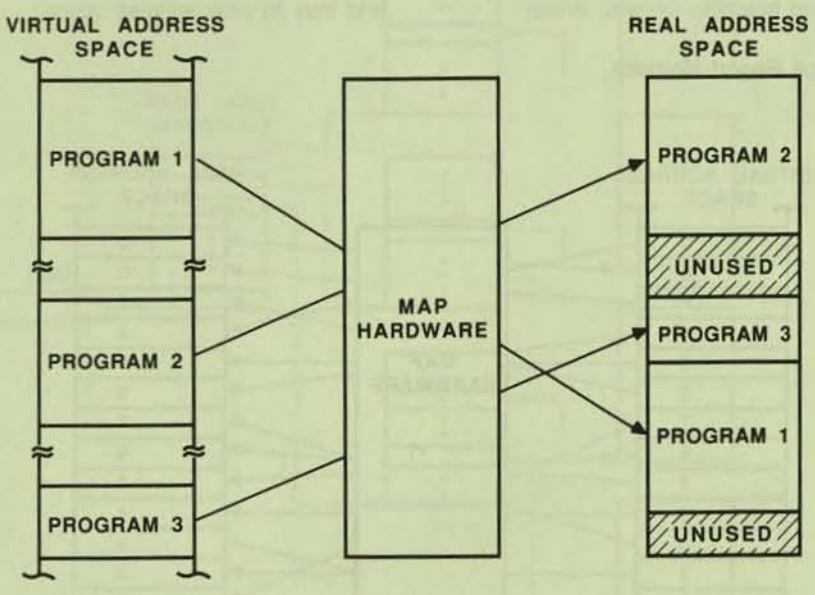


Figure 3.3 A Simple Mapping Scheme



Most 32-bit applications will employ some form of multiprogramming (either multitasking or multiuser operation), and in these cases mapping is essential to prevent the multiple programs from interfering with each other. Mapping allows different programs, users, or processes to have their own virtual address spaces. The mapping hardware provides the mechanism for translating these independent virtual address spaces into the same physical space and for protecting each user's space from interference by another program.

For efficiency, mapping is usually done in blocks of addresses instead of independently for each virtual address. The earliest mapping systems translated the entire virtual address space of a program as a single unit (see Figure 3.3). More recent systems employ smaller blocks in order to get a more fine-grained control over memory. The most widely used systems are based on fixed-sized blocks called *pages*, which are usually some low multiple of .5K or 1K bytes in length. The CLIPPER uses a paged-based mapping scheme with 4K byte pages.

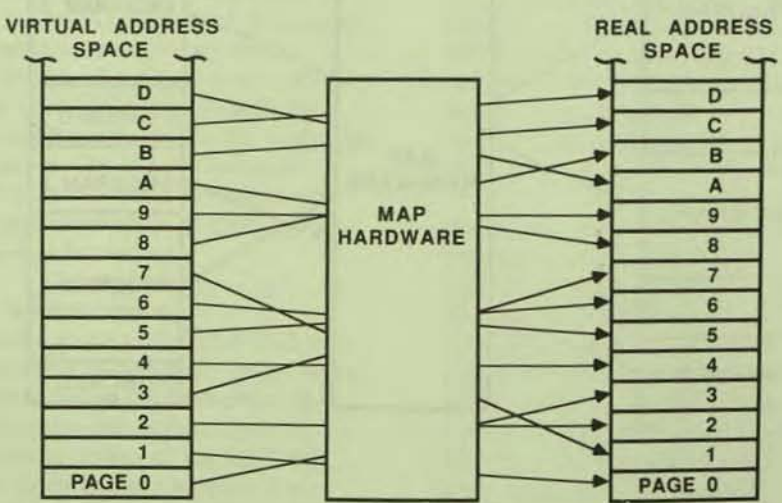
In the page-based mapping system, virtual address space is broken up into thousands of pages, each with the same size. Similarly, real address space is broken up into pieces, also called pages, having this same size. The mapping operation associates a page in real space with each page of virtual space (see Figure 3.4). Since mapping is done in units of a page, two addresses close to each other in the same virtual page will also be close in the same real page, but two contiguous virtual pages may not be contiguous in real memory.

3.3.1 CLIPPER Two-Level, Page-Based Mapping

The CLIPPER's virtual address space of 4 gigabytes is divided into 1 million pages, each with 4K bytes. The real address space is similarly divided into 4K-byte pages.

When mapping is enabled (the M bit in the SSW is set), the CAMMU translates virtual addresses generated by the CPU into real addresses. The translation process is accomplished with a two-level hierarchy of *page tables* shown in Figure 3.5. Each process has its own collection of page tables that define its virtual-to-real address map and thus its own address space.

Figure 3.4 Page Based Memory



The base of the hierarchy is the *page table directory* (see Figure 3.6); it is one page long and contains 1024 32-bit entries, each of which can point to a page table Page tables themselves are also one page long and contain pointers to pages.

Each CAMMU contains two *Page Directory Origin* (PDO) registers; one points to the base of the page table directory of the supervisor mode program (the operating system), the other points

to the base of the page table directory for the currently executing process. Upon a process swap, the OS can simply change the user PDO to obtain a new user address space. Since there are separate CAMMUs for instructions and data, each with its own PDO, there are actually four memory maps, and thus four address spaces, active at any one time:

- supervisor instruction space
- supervisor data space
- user instruction space
- user data space

Figure 3.7 shows in more detail how the address translation process works. The operating system has programmed the PDO register to point to the base of the appropriate page table directory. When the CPU sends a virtual address to the CAMMU, the upper ten bits of the virtual address are used as an index into the current page table directory. The selected entry contains the real address of a page table. The middle ten bits of

Figure 3.5 Two-Level Address Translation

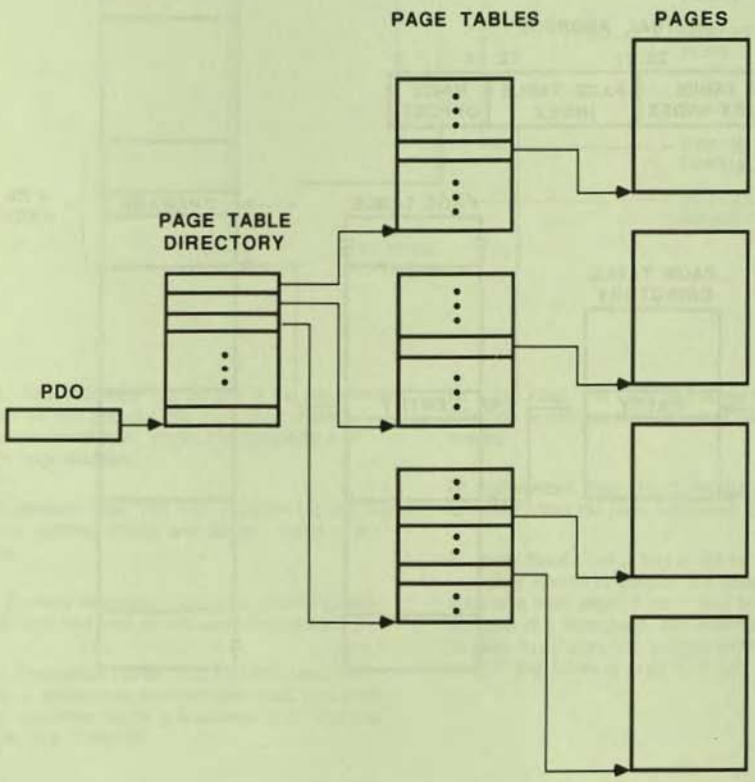
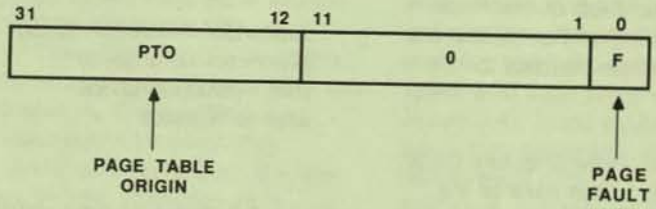


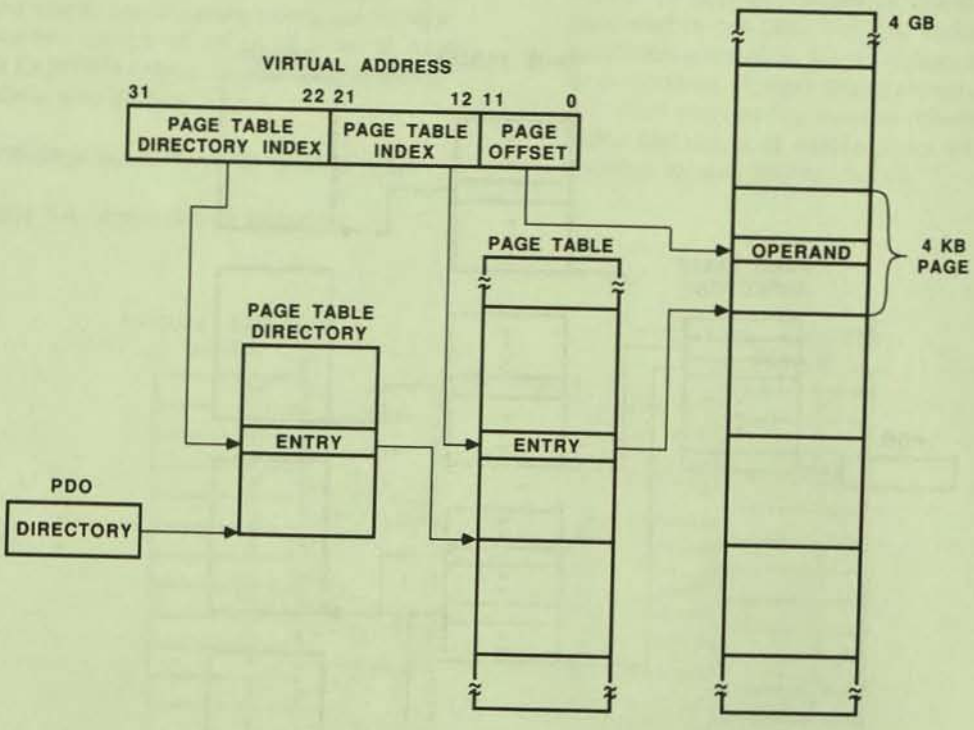
Figure 3.6 Page Table Directory Format and Description



PTO: Page Table Origin This 20-bit field contains the base address of the page table to be used for address translation.

F: Page Fault The F flag is set or cleared by the operating system to indicate the absence or presence of a valid page table pointed to by the PTO field in the entry. A set F flag indicates absence of a valid page table, and attempted DTU address translation with a page table directory entry having a set F flag forces a page fault trap.

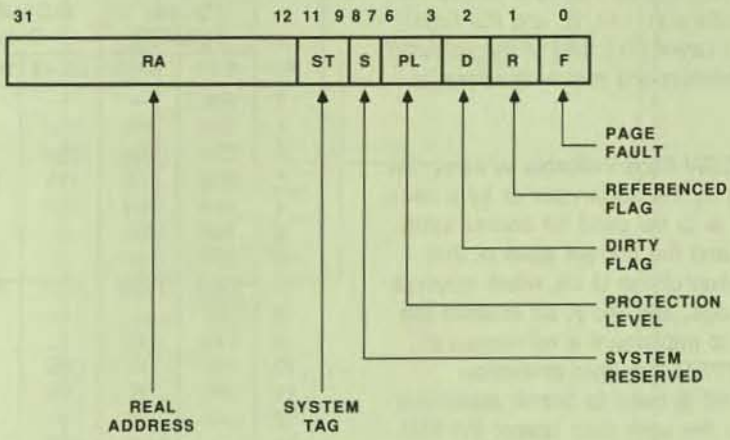
Figure 3.7 CLIPPER Address Translation



the virtual address are used as an index into this page table. The entry selected this time contains the real address of a page. Concatenating the lower 12 bits of the virtual address to the real address of the page produces the 32-bit real address of the operand. To save the overhead of page table lookups every time an address is sent from the CPU, each CAMMU caches address translation information on 128 frequently used pages in a Translation Lookaside Buffer (TLB). This TLB is searched concurrently with cache accesses in the CAMMU; only when mapping data for the requested page is not in the TLB does an access to a memory-resident directory or page table have to be made. The CLIPPER TLB has a hit ratio in excess of 99%. (Refer to Section 3.5 for more details on the TLB.)

Other bits in the page table entry are useful for virtual memory, as described in the next section. As mentioned above, at any one time the CLIPPER has four virtual address spaces (supervisor

Figure 3.8 Page Table Entry Format and Description



RA: Real Address The 20 bits of the RA field are used as real address bits <31:12> following address translation. These bits constitute a 4K byte page address.

ST: System Tag This field identifies the caching policy, caching space, and address space of the page.

S: System Reserved This is a general-purpose 2-bit field reserved for the operating system.

PL: Protection Level The CAMMU uses this field to determined whether data read, data write, and instruction fetching is allowed to or from the page. See Table 3.1.

D: Dirty Flag The Dirty flag is set by the CAMMU to indicate that the page has been altered.

R: Referenced Flag The CAMMU sets the R flag to indicate that the page has been accessed.

F: Page Fault The F flag is set or cleared by the operating system to indicate the absence or presence of a valid page. A set F flag indicates absence of a valid page, and attempted DTU address translation with a page table entry having a set F flag forces a page fault trap.

data and instruction, user data and instruction) and three real address spaces (main memory, boot, and I/O). The PDOs in each CAMMU determine the maps that define the two supervisor and two user virtual spaces. The page table entries (see Figure 3.8) contain the system tags that define which of the three real address spaces is accessed. When a virtual address is translated, this tag value is put on the system bus along with the translated real address.

The table entry also contains protection information about the page. For example, pages can be set to be read-only or execute only, or to limit access to the supervisor. Access checking for each page is performed in hardware and is determined by a protection matrix that is a function of the state of the SSW's U, UU, K, and KU flags and the Protection Level (PL) field in the selected table entry. The permissions matrix is shown in Table 3.1.

The state of the SSW flags indicates whether the memory access is by the supervisor or by a user, which protect key is to be used for access verification (K or KU), and the current state of that key. The combination of the U bit, which governs user/supervisor mode, and the K bit enables the operating system to implement a refinement of the standard CLIPPER two level protection scheme. The UU bit is used to permit supervisor mode accesses to the user data space; the KU bit is used instead of the K bit in this situation. The operating system can dynamically manage the page table entry formats and the states of the SSW flags.

Pages may be shared between processes by putting an entry for the same real page in page tables belonging to each process. The supervisor may access user pages by a similar mechanism—mapping one of its virtual pages into a real page that is also being used by a user program. The CLIPPER also provides a special mechanism that lets the supervisor use the user PDO for operand addresses; this facilitates rapid access by the supervisor to the entire user address space.

The lowest 8 pages of the supervisor virtual address space are permanently mapped, via hardwired logic in the CAMMU constituting the hardwired TLB (HTLB). Figure 3.9 shows the HTLB mapping of supervisor virtual pages to pages in real memory. Supervisor virtual pages 0–3 are translated into real pages 0–3 (main memory space); supervisor virtual pages 4–5 are

TABLE 3.1 Page Access Encoding

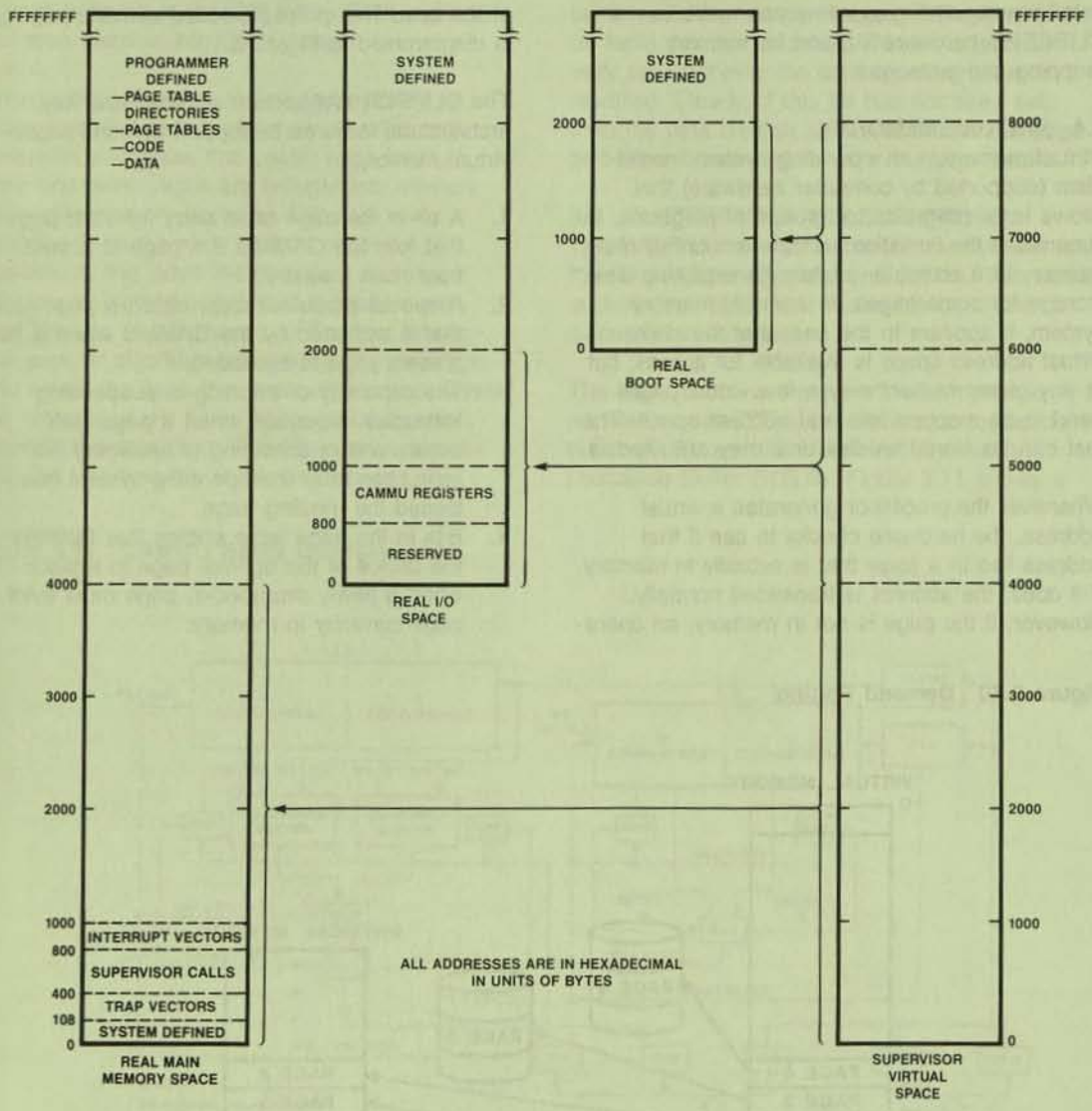
	Supervisor Mode (U=0)				User Mode (U=1)	
	UU=0 D- and I-CAMMU		UU=1 D-CAMMU Only		D- and I-CAMMU	
PL	K=1	K=0	KU=1	KU=0	K=1	K=0
0	RW	—	—	—	—	—
1	RW	RW	—	—	—	—
2	RW	RW	RW	—	RW	—
3	RW	RW	RW	RW	RW	RW
4	RW	RW	RW	R	RW	R
5	RW	RW	R	R	R	R
6	RW	R	R	R	R	R
7	RWE	RWE	RWE	RWE	RWE	RWE
8	RE	—	—	—	—	—
9	R	RE	—	—	—	—
10	R	R	RE	—	RE	—
11	R	R	RE	RE	RE	RE
12	—	RE	—	RE	—	RE
13	—	—	RE	—	RE	—
14	—	—	—	RE	—	RE
15	—	—	—	—	—	—

Notes:

—	No Access Allowed
R	Read Only Allowed
RW	Read and Write Allowed
RE	Reads and Executes Allowed
RWE	Reads, Writes, and Executes Allowed

Flags not called out in a column are "don't cares." For example, in the case where U=UU=K=0 the setting of KU is ignored.

Figure 3.9 HTLB Mapping



translated into real pages 0-1 (I/O space); and supervisor virtual pages 6–7 are translated into real pages 0-1 (Boot space). A detailed description of the HTLB is given in Section 3.5.

Chapter 9, Software Design, explains in greater detail how operating systems can make use of CLIPPER's hardware support for memory mapping and protection.

3.4 VIRTUAL MEMORY

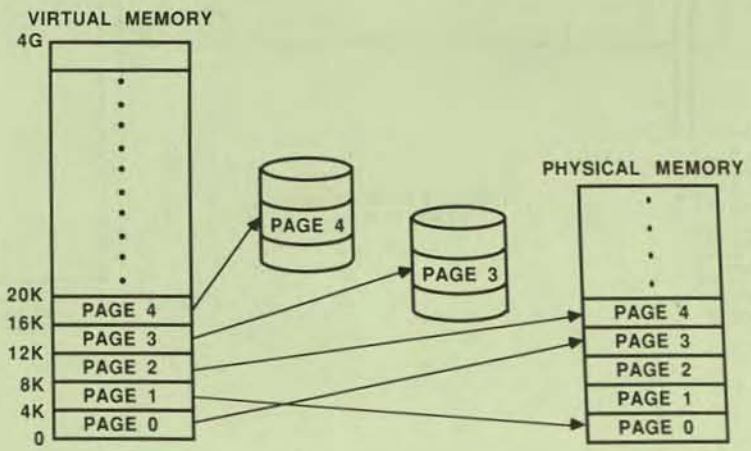
Virtual memory is an operating system mechanism (supported by computer hardware) that allows large programs, or groups of programs, to circumvent the limitations on the amount of real memory in a computer system by exploiting disk storage for some pages. In a virtual memory system, it appears to the user that the entire virtual address space is available for access; but at any given moment only a few virtual pages need to be mapped into real address space. The rest can be stored on disk until they are needed.

Whenever the processor generates a virtual address, the hardware checks to see if that address lies in a page that is actually in memory. If it does, the address is translated normally. However, if the page is not in memory, an operation called a *page swap* is performed, and the operating system loads the missing page from disk. If this swap is performed rapidly enough and if missing pages are relatively infrequent, the virtual memory system will perform nearly as well as one with far more memory, and at a fraction of the cost. This process, called *demand paging*, is diagrammed in Figure 3.10.

The CLIPPER architecture provides four key architectural features to support demand-paged virtual memory:

1. A bit in the page table entry for each page that tells the CAMMU if a page is absent from main memory.
2. A special processor trap, called a *page fault* that is activated by the CAMMU when a not-present page is accessed.
3. The capability of aborting or suspending instruction execution when a page fault occurs and re-executing or resuming the instruction after the operating system has loaded the missing page.
4. Bits in the page table entries that facilitate the choice of the optimal page to replace when a newly swapped-in page must evict a page currently in memory.

Figure 3.10 Demand Paging



A CLIPPER page table entry (see Figure 3.8) contains the page fault (F) bit which is set if the corresponding page is absent from memory. The page fault exception is one of the 18 hardware trap conditions; when it is activated, the operating system can inspect the address that caused the fault (it is stored in the CAMMU's Fault register) and then swap in the missing page.

Initially, all pages are on disk and memory is totally free. But as the program executes and generates addresses that cause page faults, more and more pages are brought into memory. Eventually memory becomes full, and newly swapped-in pages must replace pages in memory. At this point the operating system attempts to replace pages that it predicts are least likely to be referenced in the near future. One popular algorithm for making this prediction is to select the page that has not been accessed. The referenced (R) bit in the page table entry supports this algorithm. This bit is set automatically the first time the corresponding page is accessed. By periodically examining and clearing the bit, the operating system can identify pages that have not been recently used.

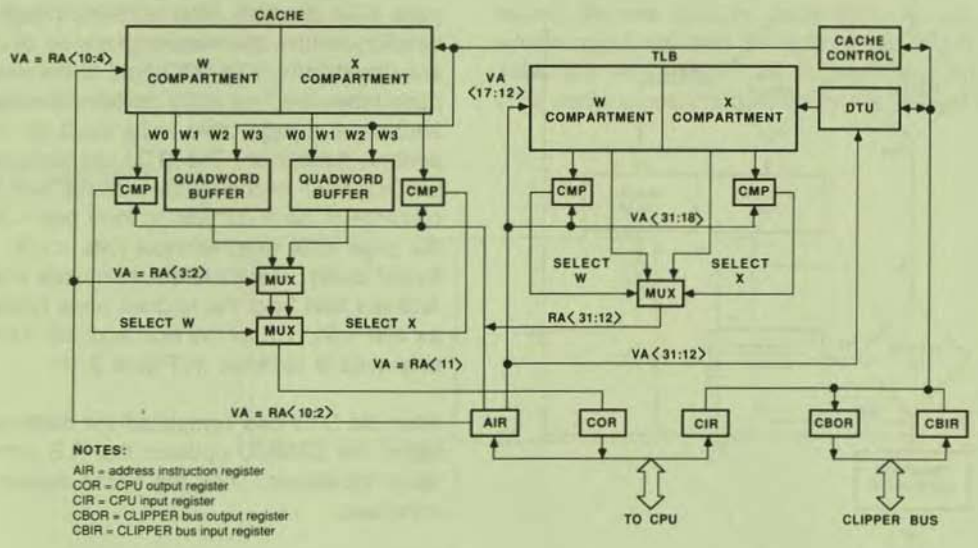
Once the operating system has selected a page to evict, it must decide whether to write the page back to disk, or simply discard it. The dirty (D) bit facilitates this decision. This bit is automatically set whenever the corresponding page is modified. Clearly, if this bit has not been set, then the data on disk already matches the page and there is no need to write the page back.

3.5 CLIPPER MMU IMPLEMENTATION

The CAMMU contains the cache mechanism and the memory management unit (MMU). The cache is described in Chapter 4; the MMU will be discussed in this section.

The MMU contains three parts—the dynamic translation unit (DTU), the translation lookaside buffer (TLB), and the Hardwired Translation Lookaside Buffer (HTLB). Figure 3.11 shows a

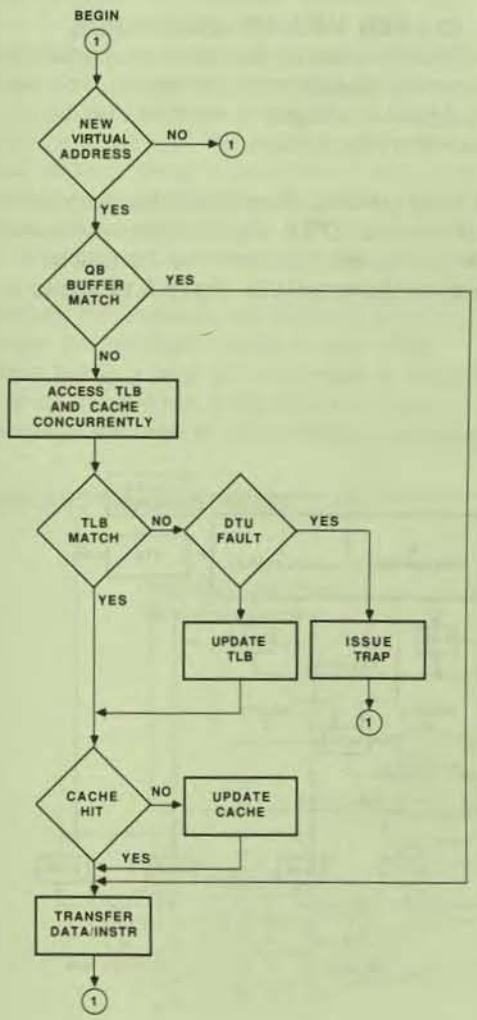
Figure 3.11 CAMMU Block Diagram



block diagram of the CAMMU; the right side comprises the MMU. The basic logic flow of CAMMU operation, including cache accesses, is shown in Figure 3.12.

The TLB is a two-way set associative cache that is used by the MMU for fast address translation requiring no access to main memory. It consists of 64 sets of lines, with each set containing a W and an X compartment line, and an associated U flag. Figure 3.13 shows an abstract picture of the TLB, and Figure 3.14 shows the format of a TLB line.

Figure 3.12 Basic CAMMU Functional Flow



TLB accesses are made concurrently with accesses to the cache (cache accesses are described in Chapter 4). For each virtual address presented to the CAMMU (see Figure 3.15), bits <17:12> of the virtual address are used to select a TLB line set at the same time bits <10:4> are selecting a cache line set. Then bits <31:18> of the virtual address are compared with the virtual address field (VA) of both lines in the set. If a match occurs, and if the appropriate access protection bits allow the data or instruction access, we have a TLB hit, and the corresponding real address field (RA) is used by the cache in its comparisons. If there is no match, we have a TLB miss, and the DTU attempts address translation using page tables in memory.

In order to translate the virtual address, the DTU concatenates bits <31:22> of the virtual address with the contents of the appropriate PDO register (user or supervisor depending on the operating mode) to form the most significant 30 bits of a page table directory entry address. (Page table directory entries are word-aligned, so bits <1:0> are forced to 0). The PTO field of the selected page table directory entry contains the base address of a page table to be used for the address translation. The DTU unit concatenates bits <21:12> of the virtual address with the contents of the PTO field to form bits <31:2> of the page table word address (bits <1:0> are forced clear). The CAMMU then uses the Real Address field from the fetched page table entry as bits <31:12> of the real address. This sequence is depicted in Figure 3.16.

After the DTU has completed the address translation, the CAMMU updates the TLB with this latest translation. Then the cache access continues.

3.5.1 HTLB

As discussed in Section 3.3, the CAMMUs include hardwired TLB lines which ensure TLB

TABLE 3.2 HTLB Virtual Address Translations

Virtual Supervisor Page Number	Real Page Number	Real Address Space	Page Assignment	Protection Level (U=0)	System Tag	
					ST, TG	Description
0	0	Main Memory	General-Purpose and Interrupt and Trap Vectors	RWE	1	shared, write through, main memory space
1	1	Main Memory	General-Purpose	RWE	2	private, copyback, main memory space
2	2	Main Memory	General-Purpose	RWE	3	noncacheable, main memory space
3	3	Main Memory	General-Purpose	RWE	3	noncacheable, main memory space
4	0	I/O	I/O, CAMMUS and Reserved	RW	4	noncacheable, I/O space
5	1	I/O	I/O	RW	4	noncacheable, I/O space
6	0	Boot	General-Purpose	RWE	5	noncacheable, Boot space
7	1	Boot	General-Purpose	RWE	5	noncacheable, Boot space

hits in both mapped and unmapped mode of supervisor virtual pages 0–7. With the exception of CAMMU I/O space (the upper half of page 0 of I/O space), these pages can also be mapped through the writable TLB.

Lines in the HTLB contain the functional equivalent of the VA, RA, ST and PL fields found in the writable TLB. However, equivalents to other TLB fields are not used in the HTLB: the HTLB can be accessed only during CPU supervisor mode, so UV and SV flags are not required; HTLB lines cannot be replaced, so no Used flags are required; and pages referenced by the HTLB are dedicated pages not subject to general replacement by the operating system, so no Referenced or Dirty flags are required. When a CAMMU detects a supervisor page 0–7 virtual address, it selects the real address, protection level and system tag fields from the appropriate HTLB line. Table 3.2 shows the HTLB page assignments. (The caching policies defined by the system tag

Figure 3.13 CLIPPER TLB

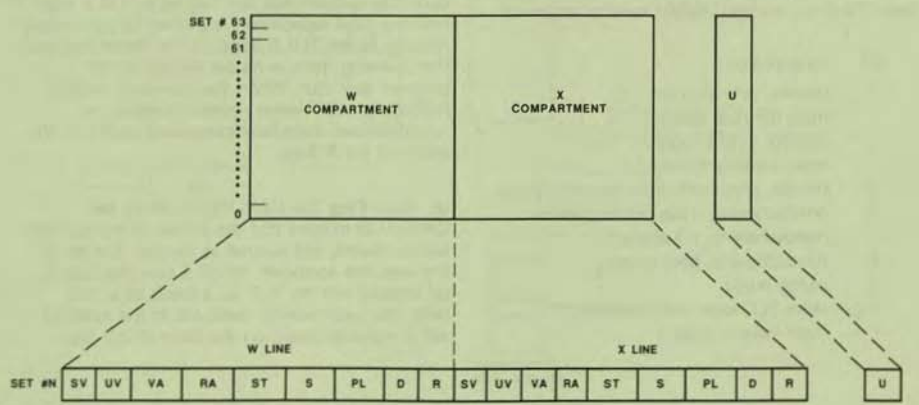
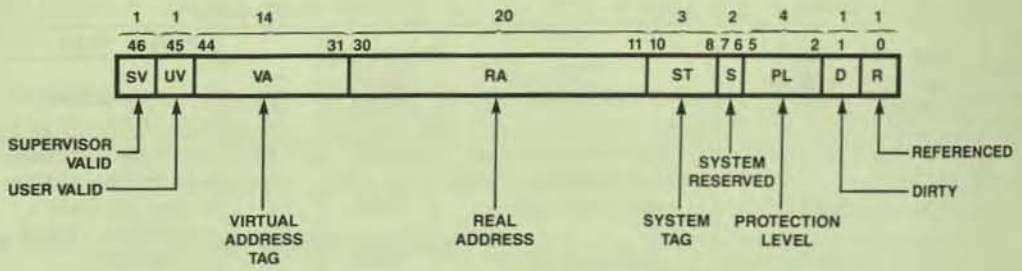


Figure 3.14 TLB Line Format and Description



SV: Supervisor Valid The SV flag when set indicates that the TLB line is used for address translation only during supervisor mode operation. All SV flags can be cleared as a group by writing to the CAMMU Reset Register and by CLIPPER hardware reset.

UV: User Valid The UV flag when set indicates that the TLB line is used for address translation only during user mode operation. All TLB UV flags can be cleared as a group by writing to the CAMMU Reset Register and by CLIPPER Module hardware reset.

VA: Virtual Address Tag This 14-bit field is used for W or X line selection once TLB set selection is complete. The VA Tag of each line is compared with CPU virtual address bits <31:18>. If there is a match, the matching line is used for the address translation.

RA: Real Address This 20-bit field is used as real address bits <31:12> once the TLB line has been matched, and access protection and validation checks have been completed.

ST: System Tag This three-bit field identifies the caching type, caching policy, and address space associated with the page referenced by the TLB line. The system tag is asserted on CLIPPER Bus lines TG<2:0> during CAMMU external accesses.

ST	Description
0	private, write-through, main memory space
1	shared, write-through, main memory space
2	private, copy-back, main memory space
3	noncacheable, main memory space
4	noncacheable, I/O space
5	noncacheable, boot space
6	cache purge
7	slave I/O mode, noncacheable, main memory space

S: System Reserved This is a general-purpose, two-bit field reserved for use by the operating system.

PL: Protection Level Associated with each virtual address is a function code asserted by the CPU which identifies each memory reference as a read, write, or instruction fetch operation, and indicates the states of the U, UU, K and KU flags in the CPU SSW. The SSW flags indicate whether the memory access is by the supervisor or by a user, which protect key is to be used for access verification (K or KU), and the state of the key. The CAMMU compares the function code with the 4-bit PL field of a selected TLB line to determine whether read access, write access, and instruction fetching is allowed. See Table 3.1.

D: Dirty Flag The Dirty flag is set by the CAMMU to indicate that the 4K byte page in main memory referenced by the TLB line has been altered. Typically the operating system uses this flag to determine whether the referenced data page must be copied to secondary storage (such as a hard disk) when the data in the page is replaced.

R: Referenced Flag The Referenced flag is set by the CAMMU to indicate that the page associated with the line has been accessed. Typically the operating system uses this flag as part of a main memory page replacement algorithm by periodically clearing all the TLB R flags via the Reset Register, then allowing them to be set during normal program execution. When the operating system replaces a main memory page, it selects an "unreferenced" page for replacement based on the states of the R flags.

U: Used Flag The Used flag is set by the CAMMU to indicate that the W line of the set was last accessed, and cleared to indicate that the X line was last accessed. When a new line has to be entered into the TLB as a result of a TLB miss, the least recently used line in the selected set is replaced based on the state of this flag.

Figure 3.15 CPU Virtual Address Format

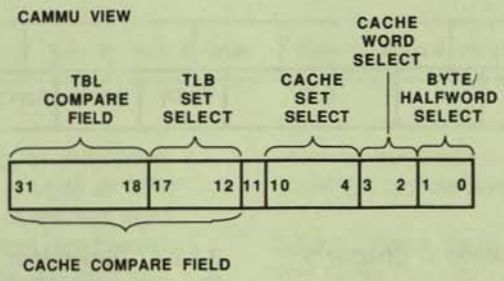


Figure 3.16 DTU Virtual Address Translation

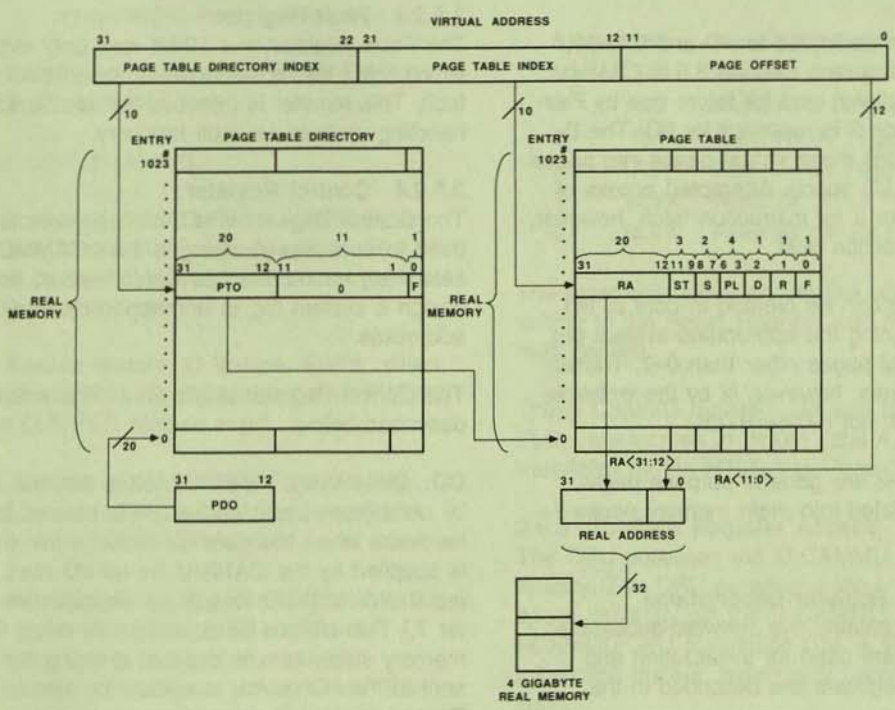
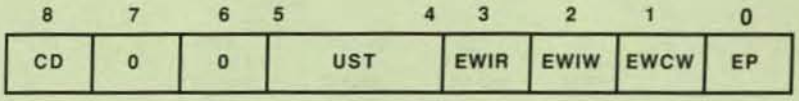


Figure 3.17 CAMMU Control Register



for each virtual page are explained in Chapter 5, Instruction and Data Caches.)

The CPU fetches interrupt and trap vectors from supervisor virtual page 0. The CAMMUs translate this page into page 0 of main memory, where these vectors must be located.

Following CLIPPER reset, the CPU fetches initial boot code from virtual page 6, which the CAMMUs translate into page 0 of boot space. The CAMMUs also translate virtual page 7 into boot space (page 1) to allow a total of 8K bytes of HTLB-translated boot code addresses.

Virtual page 4 is dedicated to I/O and CAMMU register addressing (see Section 3.5.2, CAMMU Register Descriptions) and for future use by Fairchild. Virtual page 5 is reserved for I/O. The D-CAMMU translates these virtual pages into pages 0 and 1 of real I/O space. Attempted access of virtual pages 4 or 5 for instruction fetch, however, results in a protection fault.

Additional pages can be created in boot or I/O spaces by assigning the appropriate system tag (5 or 4) to virtual pages other than 0–7. Translation of these pages, however, is by the writeable TLB or the DTU, not by the HTLB.

Virtual pages 1–3 are general purpose pages which are translated into main memory pages 1–3 by the HTLB.

3.5.2 CAMMU Register Descriptions

Each CAMMU contains five software-accessible registers which are used for initialization and control. These registers are described in the following sections.

3.5.2.1 Supervisor PDO Register

The Supervisor PDO (Page Directory Origin) Register is a 20-bit read/write register that holds the base address of a page table directory address which is used by the DTU during supervisor mode address translation.

3.5.2.2 User PDO Register

The User PDO (Page Directory Origin) Register is a 20-bit read/write register that holds the base address of a page table directory address which is used by the DTU during user mode address translation.

3.5.2.3 Fault Register

The Fault Register is a 32-bit read only register which holds the virtual address associated with a fault. This register is intended for use by trap handling routines for fault recovery.

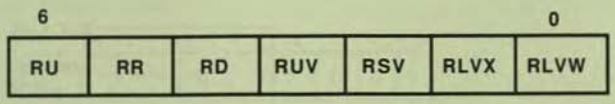
3.5.2.4 Control Register

The Control Register is a 9-bit read/write register used to enable prefetching in the I-CAMMU, to selectively enable the bus watch modes, and to assign a system tag to unmapped memory addresses.

The Control Register is shown in Figure 3.17 and described below.

CD: Clear Dirty. When this bit is set, the dirty bit on a (copy-back) cache line is cleared by hardware when the data contained within that line is supplied by the CAMMU for an I/O read (as a result of Watch I/O Reads, as explained in Chapter 7.) This bit can be so used only when the memory subsystem is capable of using the data sent to the I/O device to update its own contents. This bit is cleared on reset.

Figure 3.18 CAMMU Reset Register



UST: Unmapped System Tag. In unmapped mode (the M flag in the SSW is zero) all CPU addresses, with the exception of the first eight supervisor virtual page addresses, are treated by the CAMMUs as real addresses requiring no translation. These unmapped addresses therefore have no TLB or page table source of system tags. The CAMMUs use the two bit UST field to tag pages referenced with unmapped addresses as follows:

UST	Operation Type
0	private, write-through, main memory space
1	shared, write-through, main memory space
2	private, copy-back, main memory space
3	noncacheable, main memory space

The UST bits map to TG<1:0> CLIPPER Bus lines. TG<2> is forced to 0.

On reset, UST is set to 3.

EWIR: Enable Watch I/O Reads. EWIR, when set, enables Watch I/O Reads operation. This bit is ignored by the I-CAMMU. EWIR is set by CLIPPER Module reset.

EWIW: Enable Watch I/O Writes. EWIW, when set, enables Watch I/O Writes operation. EWIW is set by CLIPPER Module reset.

EWCW: Enable Watch CPU Writes. EWCW, when set, enables Watch CPU Writes operation. EWCW is set by CLIPPER Module reset.

EP: Enable Prefetch. EP, when set, enables I-CAMMU prefetching. When EP is clear, I-CAMMU prefetching is disabled, and the I-CAMMU fetches instructions "on demand." The state of this bit is ignored by the D-CAMMU, which always fetches on demand. EP is set by CLIPPER Module reset.

3.5.2.5 Reset Register

The Reset Register is a 7-bit write-only register that allows selective resetting of flags of the CAMMU cache and TLB (see Figure 3.18). The cache LV (Line Valid) and U (Used) flags, and the TLB SV (Supervisor Valid), UV (User Valid), D (Dirty) and R (Referenced) flags can all be selectively cleared.

The Reset Register bits and their associated reset operations are as follows:

Bit No.	Bit Name	Reset Operation
6	RU	Reset All U Flags in Cache
5	RR	Reset All R Flags in TLB
4	RD	Reset All D Flags in TLB
3	RUV	Reset All UV Flags in TLB
2	RSV	Reset All SV Flags in TLB
1	RLVX	Reset All "X" LV Flags in Cache
0	RLVW	Reset All "W" LV Flags in Cache

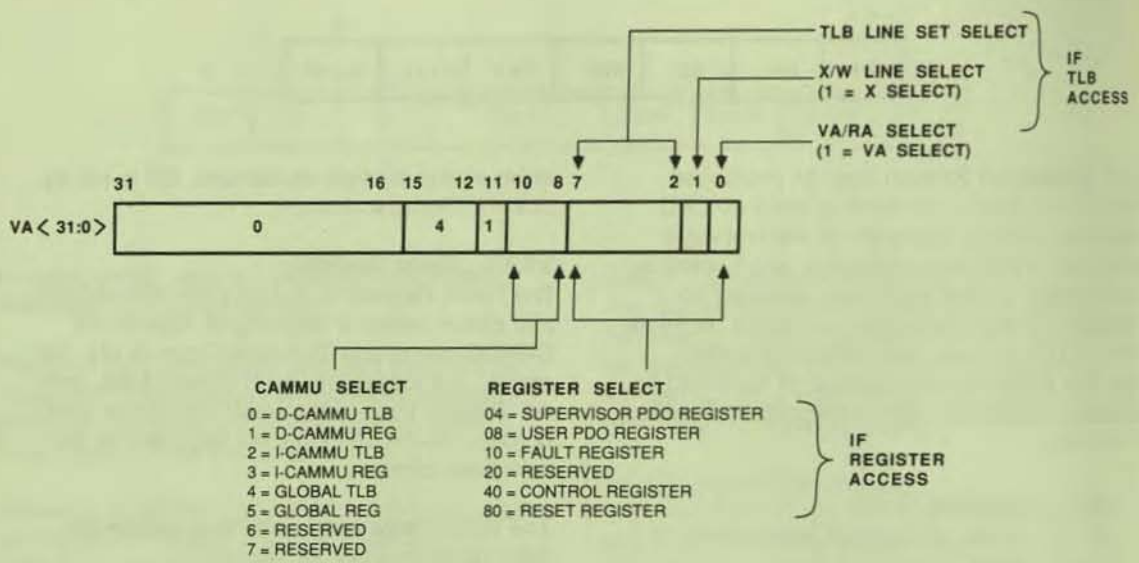
The reset operations shown are performed by writing to the Reset Register with the appropriate data pattern.

These CAMMU registers, as well as the CAMMU TLBs, are located in virtual page 4, which is translated by the HTLB into Page 0 of I/O space.

3.5.3 CAMMU Register Access

The CPU accesses the D-CAMMU I/O space directly. The CPU accessess the I-CAMMU I/O space indirectly via the D-CAMMU, because the I-CAMMU/CPU Instruction Bus is tied to CPU instruction buffers, which only transfer instructions.

Figure 3.19 CAMMU Addressing



The D-CAMMU registers are located in supervisor virtual addresses 4800–49FF (hex). The I-CAMMU registers are located in supervisor virtual addresses 4A00–4BFF. These addresses are used to access registers in individual CAMMUs.

The CAMMUs can also be addressed as a group using global addresses for TLB writes, register writes and TLB/cache reset. In systems utilizing multiple CLIPPER Modules, for example, a CPU can execute global writes to CAMMUs other than its companion D-CAMMU by accessing virtual address locations 4Cnn (hex, TLB write), and 4Dnn (hex, register write and TLB/cache reset). I/O devices can execute the global writes by accessing Cnn and Dnn (hex).

Global writes are typically used in multi-CPU systems when main memory pages that are shared by more than one CLIPPER Module are replaced. If the virtual address of a page being replaced is identical for all modules sharing the page, a single global write to CAMMU I/O space can be used to invalidate the TLB entry corresponding to the outgoing page in all CAMMUs.

3.5.3.1 CAMMU Register Addressing

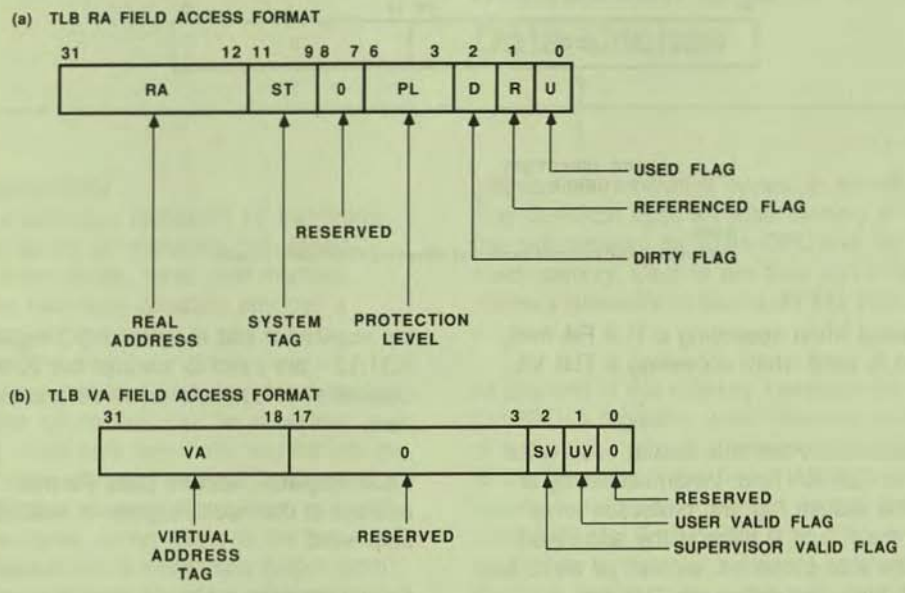
CAMMU I/O space addresses are shown in Figure 3.19. Virtual address bits <31:11> comprise the CAMMU Base Address field, which must point to the upper half of virtual page 4 for CAMMU access.

Virtual address bits <10:8> comprise the CAMMU Select field. This field identifies the following:

Bit #			CAMMU	
10	9	8	Selected	Operation/Access
0	0	0	D-CAMMU	R/W TLB
0	0	1	D-CAMMU	R/W Registers; Reset TLB/Cache
0	1	0	I-CAMMU	R/W TLB
0	1	1	I-CAMMU	R/W Registers; Reset TLB/Cache
1	0	0	Global	Write TLB
1	0	1	Global	Write Registers; Reset TLB/Cache

Note:
The TLBs and caches are reset by writing to the Reset Register.

Figure 3.20 TLB Access Data Formats



The first four entries show individual I- and D-CAMMU addressing. The last two entries show global addressing, intended for use in systems utilizing more than one CLIPPER Module. A CPU uses global addressing in such a system to access a specific register or to reset the TLB and cache in all CAMMUs in the system other than its own D-CAMMU.

Bits <7:0> of the virtual address comprise the Register Select field. This field identifies the register or the TLB field being accessed. All five CAMMU registers, and individual VA (Virtual Address) tag and RA (Real Address) fields of the TLB can be addressed.

If the operation is a TLB access, virtual address bits <7:2> address one of the 64 TLB entries, bit <1> addresses the W or X line of the TLB entry, and bit <0> addresses the VA tag or RA field of the addressed TLB line.

If the operation is a register access, virtual address bits <7:0> address the registers as follows:

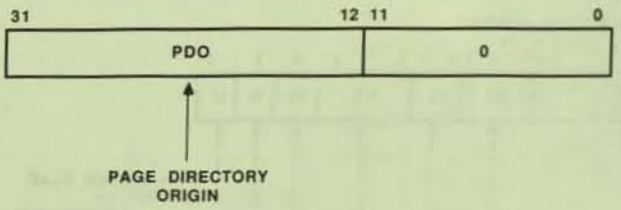
Bits <7:0>	Register Addressed
0000 0000	Reserved
0000 0001	Reserved
0000 0010	Reserved
0000 0100	Supervisor PDO (read/write)
0000 1000	User PDO (read/write)
0001 0000	Fault Register (read only)
0010 0000	Reserved
0100 0000	Control Regsiter (read/write)
1000 0000	Reset Register (write only)

3.5.3.2 CAMMU Data Format

The format of data written to and read from the CAMMUs varies according to the register or TLB field addressed. Both the fields and the number of data bits used in the 32-bit data words written to the CAMMUs differ to accomodate individual CAMMU registers and register types.

TLB Access Data Format. The two TLB transfer data formats are shown in Figure 3.20. One

Figure 3.21 PDO Register Access Format



NOTE:
This format is used for both the user and supervisor PDO register access.

format is used when accessing a TLB RA field; the second is used when accessing a TLB VA tag field.

Figure 3.20a shows the data format used when accessing a TLB RA field. When accessing an RA field, the system tag and protection level fields and the R and D flags of the addressed TLB line are also accessed, as well as the U flag of the TLB entry containing the TLB line.

Figure 3.20b shows the data format used when accessing a TLB VA tag field. The UV and SV flags of the TLB line are accessed, as well as the VA field. Note that several data bits are not used. These bits are reserved and must be zero.

PDO Register Access Data Format. Figure 3.21 shows the data format used when accessing both the supervisor and the user PDO registers. Bits <31:12> are used to transfer the 20-bit PDO data; bits <11:0> are reserved and must be zero.

Fault Register Access Data Format. The 32-bit address in the Fault Register is read as a 32-bit data word.

Control Register Access Data Format. The least-significant nine bits are used when accessing the Control Register; bits <31:9> are reserved and must be zero.

Reset Register Access Data Format. The seven least-significant bits are used when accessing the Reset Register; bits <31:7> are reserved and must be zero.

Chapter 4
Instruction and Data Caches

4.1 INTRODUCTION

Caching is a technique pioneered by mainframe computer designers for matching high-speed CPUs with lower speed, lower cost memory systems. The technique basically employs a small, high-speed buffer that is positioned between the CPU and the main memory. Frequently used data and instructions are loaded into this buffer where they can be accessed quickly; only when new items are fetched into the buffer does main memory have to be accessed. Thus the *effective memory access time* is greatly reduced. The name *cache* refers to the fact that the buffer mechanism is essentially hidden from the user, who is aware only of an apparently higher speed main memory.

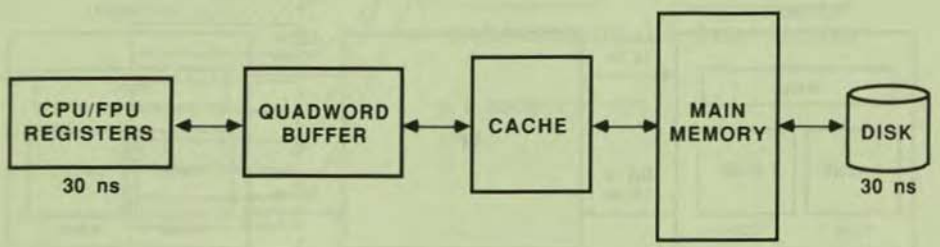
Since the CLIPPER Clock runs at 33.3 ns, the CLIPPER CPU is capable of completing a single memory access in 30 ns. It is of course possible to build a memory system using high-speed static RAMs (SRAMs), but most microprocessor systems use slower speed and much less expensive dynamic RAMs (DRAMs). The typical access time for currently available DRAMs is around 120 ns, with a conservative memory bus design bringing the total memory access time, including address and arbitration cycles, to around 500 ns. The CLIPPER uses a cache memory to bridge the gap between its 30 ns CPU and the 500 ns main memory. Caches are thus part of the *memory hierarchy* of the CLIPPER. (See Figure 4.1.)

At one end of the memory hierarchy are the CLIPPER's registers, which have an access time of one clock cycle, or 30 ns. At the opposite end of the hierarchy is the mass storage system— usually a hard disk— with an access time of around 30 ms. Bridging the gap of six orders of magnitude (a factor of a million) between the registers and the disk is the main memory and caches. The CLIPPER actually employs two levels of caches: an 8K byte main cache and a very high-speed cache-within-the-cache called a *quadword buffer.*

The effectiveness of a cache system depends on three factors:

1. cache access time
2. hit rate
3. replace time on a cache miss

Figure 4.1 Memory Hierarchy



The *cache access time* is the number of clock cycles required to access data or instructions contained in the cache. The *hit rate* refers to the frequency that the desired data or instruction is actually in the cache. If the data is found, the access is said to have produced a *cache hit*; but if the requested data is not in the cache, a *cache miss* has occurred. The hit rate is basically the percentage of cache hits to total memory accesses. Finally, the *replace time* is the number of clock cycles required to fetch an item from main memory into the cache when a cache miss occurs.

The *effective access time* is the average time to access data or instructions. It depends on the interaction of the three factors mentioned above and the character of the program itself. Most programs tend to re-access memory locations soon after they are accessed for the first time, or else to access nearby locations. This behavior is called *locality of reference*. The more locality a program exhibits, the more effective caching is at reducing effective memory access time. For typical programs, the CLIPPER caches reduce the effective access time from 500 ns, to around 100 ns, a 500% improvement in performance.

4.2 CACHE ACCESS TIME

The CLIPPER's cache memory actually consists of two 4K byte caches: the instruction cache and the data cache, each located on a CAMMU. Each CAMMU also contains a 16 byte quadword buffer. The data in the caches is organized as 256 *lines*, with each line containing 16 bytes of data. Whenever a cache hit occurs, the entire line containing the accessed item is loaded into the quadword buffer. Subsequent accesses to the same line will not require that the cache be accessed; instead, the request will be satisfied from the quadword buffer.

The quadword buffer can be accessed in one clock cycle (30 ns). If the quadword buffer misses, the cache can be accessed in an additional two clock cycles, for an access time of 3 clocks or 90 ns. The time to transfer the address over the bus to the CAMMU is 15 ns. (The CPU and CAMMU are designed to use one half of a clock cycle for this transfer.) Similarly, it takes only 15 ns to transfer the data back to the CPU. Thus the total access time for the quadword buffer is 60 ns and for the main cache, 120 ns. Figure 4.2 illustrates the CLIPPER cache access times.

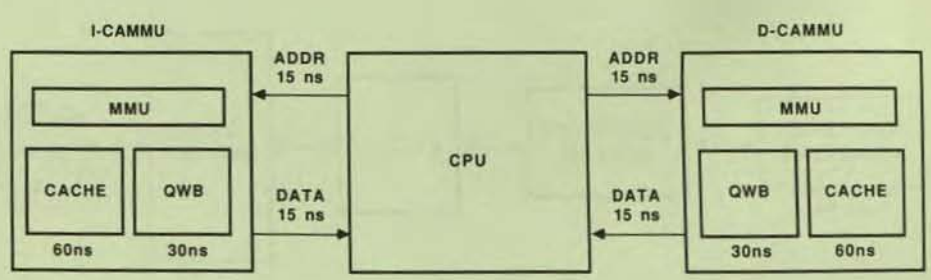
4.3 HIT RATE

The hit rate is a function of four factors:

1. the size of the cache in bytes
2. the number of bytes in a line
3. the degree of *set associativity*
4. the program's locality

Since computer architects have no control over program locality, this factor will not be discussed further, except to observe that modern structured programming techniques are very beneficial in this regard.

Figure 4.2 CLIPPER Cache Access Time

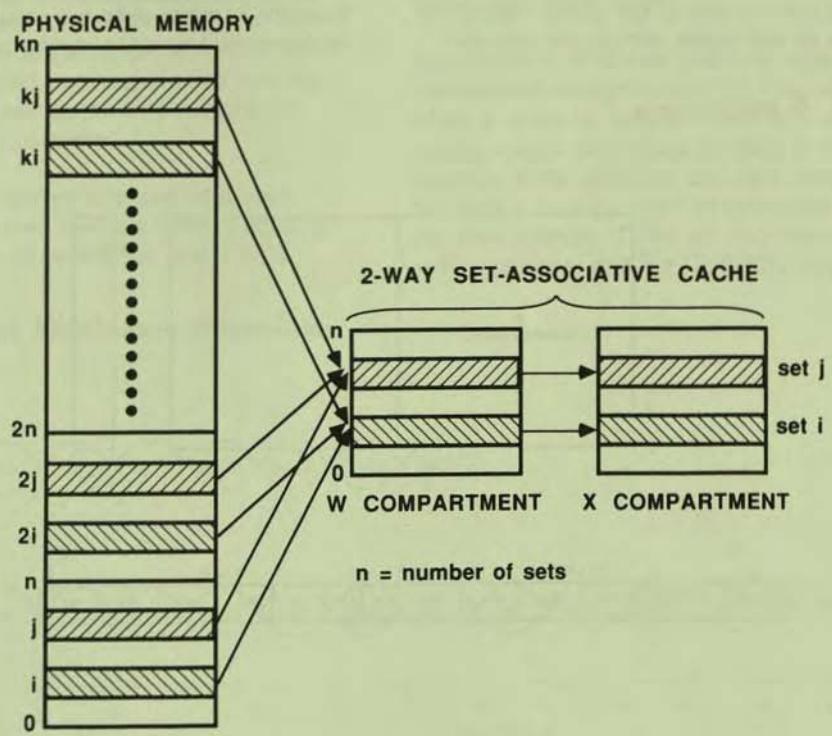


The *line size* is the number of bytes that are fetched into the cache when a memory reference is made. The more bytes that are fetched, the fewer main memory accesses have to be made, if programs have reasonable locality, though each access takes longer. In general, increasing the line size improves the effective memory access time up to a point of diminishing returns. The CLIPPER uses a 16 byte line size, which means that 16 bytes are updated when a cache miss occurs, and 16 bytes are transferred whenever main memory is accessed. Studies indicate that a 16 byte line is optimal. Each cache contains 256 lines.

If the cache is full and a cache miss occurs, the new line will have to overwrite a line of data already in the cache. The *set associativity* of the cache determines how much flexibility the cache has in determining which line to overwrite. The simplest kind of cache is a *direct mapped* cache, in which each main memory location can be written into only one cache location. A *two way set associative* cache can write each memory location into two cache locations. Similarly, four cache locations are available in four way set associativity, and so on. The higher the degree of associativity, the more flexible the cache and the less likely useful data is overwritten. Figure 4.3 illustrates the CLIPPER's two way set associativity.

In the CLIPPER cache (see Figure 4.4), each set contains 2 lines of 16 data bytes, and there are 128 sets. The collection of all the left hand lines is called the *W compartment*, the collection of

Figure 4.3 Set Associativity



right hand lines is called the *X compartment.* There are a total of 2K bytes in each compartment. Physical memory is divided into 4K byte

pages, and within each page, a location is mapped into a unique location in either of the two compartments. Consequently, several main memory locations are mapped into the same pair of cache locations. The first time main memory is accessed, 16 bytes are fetched and stored in a line of the W compartment. If a new main memory access would require that that line be overwritten, the cache will instead employ the corresponding line in the X compartment. If the corresponding lines of both compartments are full, the cache will overwrite the one that was least recently used, using the U bit. The U bit is automatically set when the W compartment is accessed and cleared when the X compartment is accessed.

Thus we can see that the efficiency of a cache depends on both the line size and set associativity, and not just the size of the cache. Figure 4.5 shows how hit rate varies with cache size, line size and associativity. Notice that the CLIPPER's 8K byte total cache size along with its 16 byte line size and two way set associativity yield a hit rate that is about the same as a 128K byte, 4 byte line, direct mapped cache. In fact, for typical applications, the data cache has a hit rate of around 90%, while the instruction cache's hit rate is slightly higher. And when prefetch is enabled, the instruction cache can exceed a 96% hit rate.

Prefetching is a mechanism available on the instruction CAMMU for bringing into the cache the next 16 bytes of memory before they are accessed. The CAMMU maintains a copy of the Program Counter and uses it to fetch instructions before the CPU requests them. For in-line code sequences, the prefetch mechanism can insure a 100% hit rate, since prefetch is performed concurrently with other CPU and CAMMU operations. (Prefetching is described in Chapter 5, Pipelining.)

4.4 REPLACEMENT TIME

Inevitably a cache miss will occur. The miss *replacement time* is the number of clock cycles

Figure 4.4 CLIPPER Cache

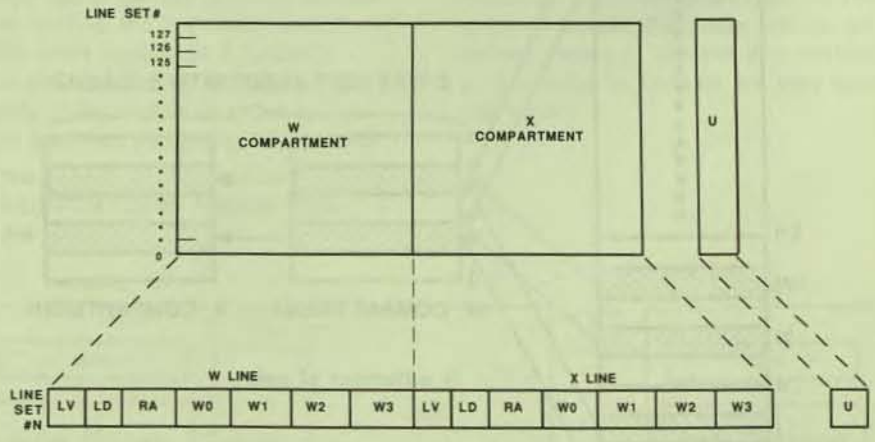
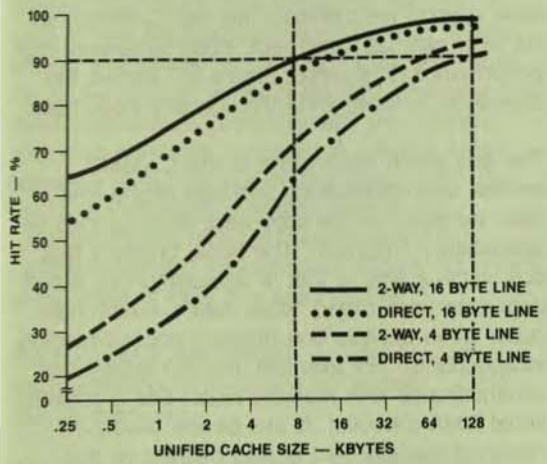


Figure 4.5 Cache Hit Rate



Source: U.C. Berkley

that occur from the time the miss is detected by the CAMMU until the last byte of the line is loaded into the cache. Leaving aside the speed of memory, this time is determined by the bus clock rate, the line length, and the number of clock cycles required to transfer a line over the memory bus, as well as the time required for CAMMU internal operations.

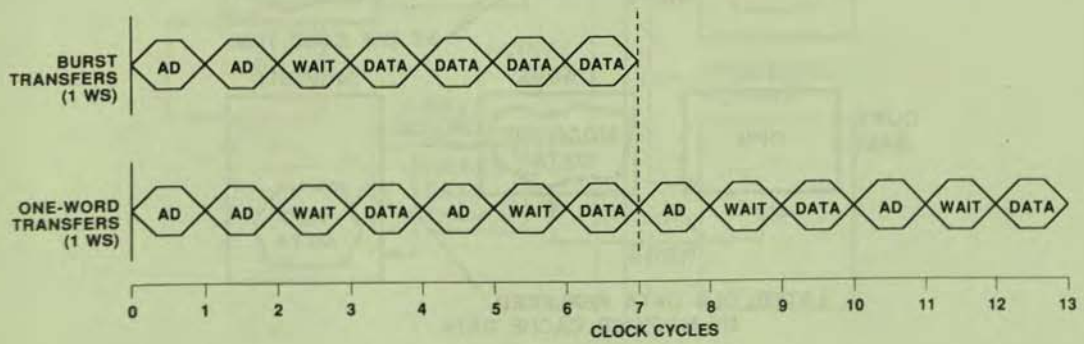
The CLIPPER's memory bus runs at a clock speed of 16.7 ns (i.e., one bus cycle is equal to two CPU cycles). Since the line length is 16 bytes, the replacement time depends on how many clock cycles it takes to transfer 16 bytes over the bus.

Conventional 32-bit microprocessors transfer one word (32 bits or 4 bytes) of data over the bus with each memory access, and the slow speed of DRAMs introduces at least one wait state for each access. So these conventional machines require at least 12 clock cycles to transfer 16 bytes (see the lower half of Figure 4.6).

By contrast, the CLIPPER employs a special high-performance technique called *burst mode* to transfer all four words (16 bytes) in a single memory access. Since only one access is made, only one address is transferred to memory (the starting address of the line of 16 bytes) and only one wait state is needed if slow speed DRAMs are used. The next four clock cycles transfer the four words of the line. Burst mode transfers of the entire line are twice as fast as conventional 4-word transfers. The top part of Figure 4.6 illustrates burst mode timing.

The concepts of hit rate and miss replacement are relatively straightforward for read accesses: when a cache hit occurs, the data is used; when a miss occurs, an access is made to main memory. Write accesses are more complicated, because a strategy must be developed for updating main memory. There are two main cache write strategies, and the CLIPPER supports them

Figure 4.6 Cache Replacement Time



both:

- write-through
- copy-back

The user may designate any page in memory as cacheable or noncacheable, and if cacheable, select one of the two caching strategies for that page.

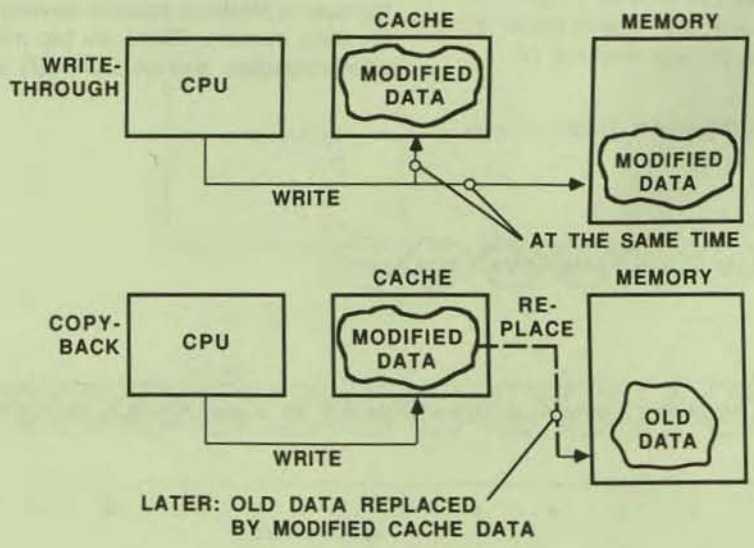
The simplest strategy is called *write-through*. Under this scheme, main memory is updated every time the cache is altered. Since the cache and main memory are changed at the same time, they always contain the same data and main memory is always "up to date". The penalty is that little or no benefit is derived from the cache for writes, and thus the data cache is only about half as effective as it might be.

With the copy-back strategy, memory is updated only when a line that has been modified in the cache needs to be overwritten by another line. At that point, the old modified line is "copied back" to memory before it is overwritten. The copy-back strategy minimizes memory accesses and thus provides the highest performance, but this strategy requires more hardware support. The designers of the CLIPPER built the needed hardware support mechanisms into the CAMMU, so the user can use copy-back mode whenever high performance is required. Figure 4.7 shows the difference between write-through and copy-back.

The *bus watch* mechanism in the CAMMU insures data consistency between cache and main memory, so the copy-back strategy can be successfully employed. The basic problem that bus watch solves is that of accessing old, out-of-date data (also called "stale data"), which can occur when multiple bus masters access memory independently. For example, the I/O subsystem communicates with memory much like a conventional DMA controller. If this device reads a region of memory that is also cached on the CAMMU using a copy-back strategy, it will access stale data if the memory is not up to date. Similarly, if the I/O subsystem writes to a cached region, the CPU will then be reading stale data in the cache.

Bus watch eliminates this problem by automati-

Figure 4.7 Cache Write Strategies



cally insuring data consistency. The bus watch mechanism fulfills read requests by other bus masters from the cache instead of memory, if there is any difference between the two. It also insures that writes by other bus masters update the cache as well as main memory. Bus watch monitoring occurs in parallel with the memory access to eliminate or minimize the bus watch operation effect on system bus throughput. Bus watch is controlled by setting flags in the CAMMU control register (see Section 3.5.2, CAMMU Register Descriptions). Figure 4.8 shows an example of how bus watch works.

The bus watch mechanism for various system bus operations is explained in Chapter 7, CLIPPER System Bus.

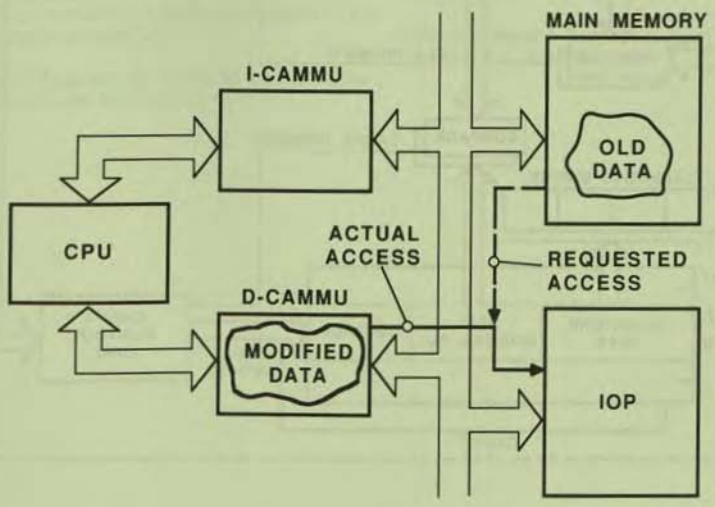
4.7 CLIPPER CACHE IMPLEMENTATION

The CLIPPER Cache is closely tied to the MMU and its TLB, which were described in Chapter 3, Memory Management. The CPU generates 32-bit virtual addresses, which are translated by the MMU/TLB into real addresses. The cache mechanism then compares these real addresses with addresses stored internally, and if a match is found, the word associated with the internal address is returned to the CPU.

In more detail, the following process occurs every time the CPU generates a virtual address.

1. If it is an instruction access, the virtual address is sent to the instruction CAMMU; a data address is sent to the data CAMMU.
2. The virtual address is compared with the value stored in a buffer called the *virtual address cache*. This buffer contains the last virtual address sent to the cache. (The quadword buffer contains the last line accessed, one of whose words was pointed to by the last virtual address.)
3. If the two values match, we have a *quadword buffer hit*, and bits <3:2> of the virtual address are used to select one of the four data words in the buffer to be returned to the CPU. (Actually, each CAMMU contains two quadword buffers, one for each compartment, but the most recently accessed one is used.)
4. If the two values do not match, the quadword buffer misses, and the CAMMU proceeds

Figure 4.8 Bus Watch



concurrently to translate the virtual address and access the cache. The virtual address is stored in the virtual address cache.

5. The MMU/TLB translates bits <31:12> of the virtual address into a 20-bit real value. Bit 11 of the virtual address is appended, yielding a 21-bit value called the *real address*. Meanwhile, bits <10:4> of the virtual address have selected one of the 128 sets in the cache.
6. The 21-bit real address from the TLB is compared with the real address tag field of both lines in the selected set of the cache.
7. If the two values match for one of the two lines, we have a *cache hit*, and again bits <3:2> of the virtual address are used to select one of the four data words to be returned to the CPU. The line itself is loaded into the quadword buffer.
8. If a cache miss occurs, the cache control mechanism causes a main memory access to be generated, by appending virtual address bits <10:0> to the 21-bit real address to produce a 32-bit real address.
9. The main memory access causes a new line to be loaded into the cache and the quadword buffer, and steps 6 and 7 are performed again.

Figure 4.9 shows a schematic representation of this process.

Figure 4.10 shows the layout of the CLIPPER cache, and describes the fields in each line.

Figure 4.9 CLIPPER Cache Access Mechanism

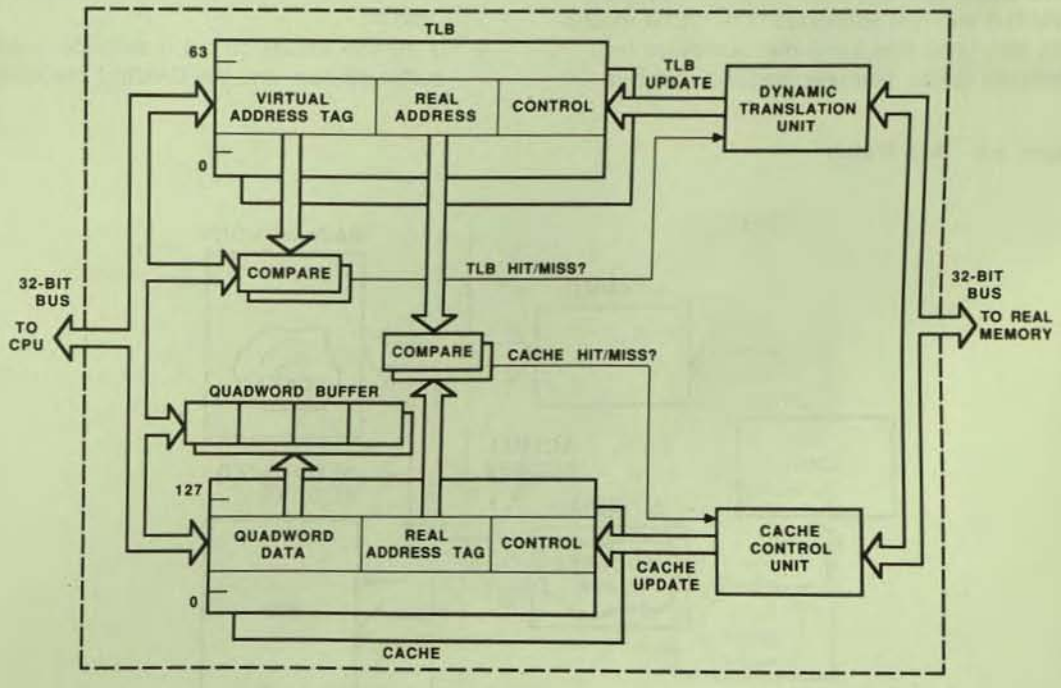
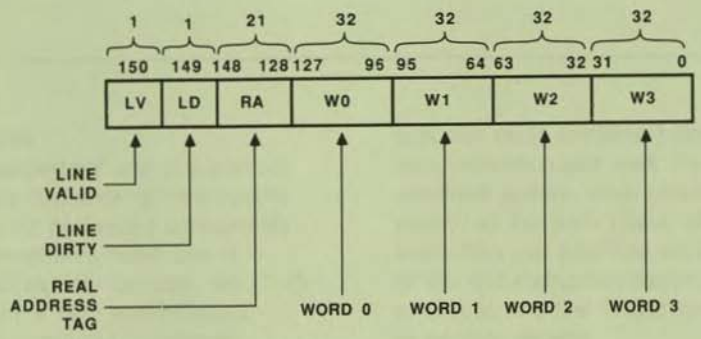


Figure 4.10 CLIPPER Cache Line Format and Description



LV: Line Valid The LV bit, when set, indicates that the data in the associated line is valid; when clear, it indicates that the data is invalid.

A line LV bit is set by the CAMMU when it loads new data into the cache line. The bit is cleared by the CAMMU, operating as a slave, in response to CLIPPER Bus activity when its bus watch modes are enabled. Specifically, this bit is cleared during a quadword write to a main memory location which matches the particular cache location. During this time the CAMMU asserts the $\overline{\text{CBSY}}$ signal on the CLIPPER Bus, preventing another bus transaction from beginning until the CAMMU has cleared this bit. If the CPU has addressed this same cache line prior to the CLIPPER Bus' write operation, the CPU has priority and the bit is not cleared until the CPU access is completed. The LV bit is also cleared by a cache purge operation, as described in Section 7.2.4, Memory Space System Tag.

All cache LV flags can be cleared as a group via the Reset Register and by hardware reset.

LD: Line Dirty The LD bit is set by the CAMMU to indicate that data in the cache line is not current with main memory. This occurs when the CAMMU is operating in the copy-back mode, and a CPU write to memory results in a cache hit. In this case, the data is written to the cache but not to main memory, resulting in "dirty" (not current) main memory data.

This bit is cleared by the CAMMU when the CD (Clear Dirty) bit in the Control register is set and the cache provides more current data for an I/O read. See Section 8.2.1.2, Clear Dirty Support.

RA: Real Address Tag The RA tag is used for cache line selection. The RA tags of both the W and X compartment lines are compared with translated address bits <31:12>. Accessed data is located in a matching line.

W0–W3: Word 0–3 Words 0–3 are the four 32-bit data words in the cache line.

Chapter 5
Pipelining

5.1 INTRODUCTION

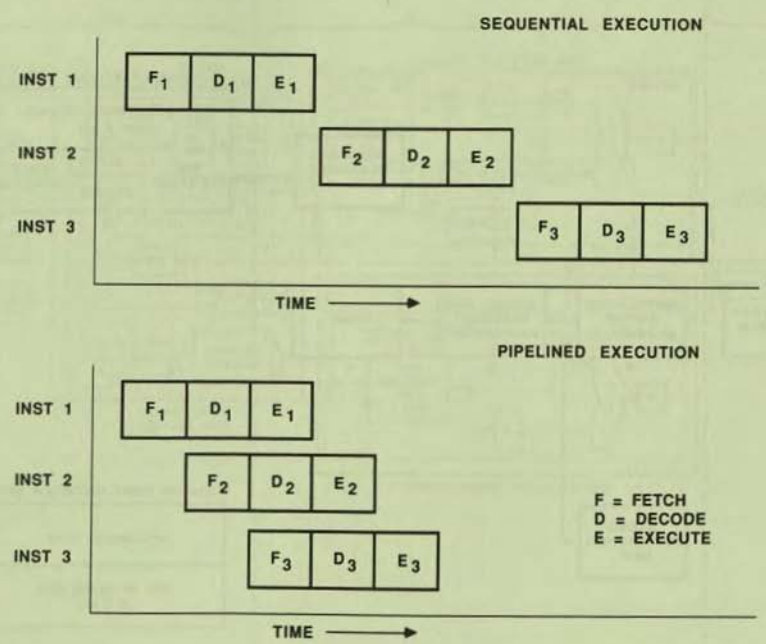
Pipelining was developed for use in supercomputers to improve performance by overlapping operations that can be performed concurrently. Most 32-bit microprocessors make use of a simple form of pipelining. By contrast, the CLIPPER pipeline system is very sophisticated, making use of supercomputer techniques.

Figure 5.1 provides an illustration of the performance benefits of even a simple form of pipelining. In non-pipelined systems, each instruction follows sequentially after the previous, and all phases of an instruction (fetch, decode, execute) must complete before any phase of the next instruction can start. By contrast, in a pipelined system, each phase proceeds independently, so the fetch phase of the second instruction can begin as soon as the fetch phase of the first instruction completes, without having to wait for the first instruction to finish its decode or execute phases.

The CLIPPER combines the benefits of this simple three-phase pipeline with extensive use of concurrency and parallelism within each phase of the pipe, especially the execute phase.

5

Figure 5.1 Pipelining



5.2 THREE-PHASE PIPELINE

The CLIPPER pipeline has three phases:

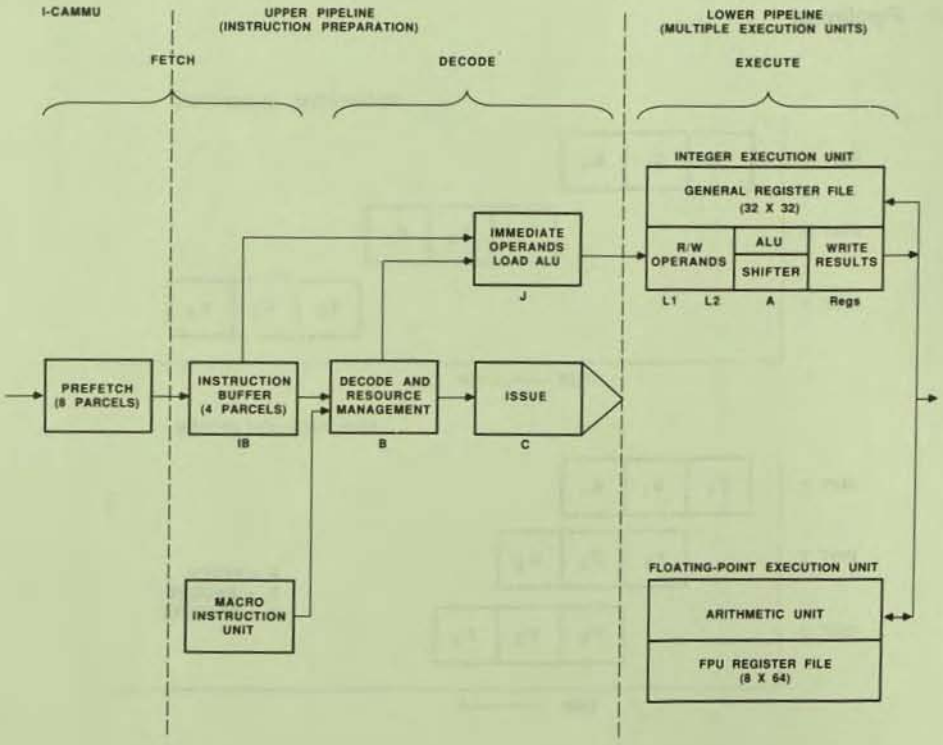
Fetch — Instructions are fetched from the instruction cache to the CPU's instruction buffer (which can hold two words, or up to four instructions), or instructions can be fetched from the Macro Instruction Unit as a result of some previous macro instruction. In addition, the instruction cache can prefetch instructions ahead of actual demand.

Decode — Instructions from either the instruction buffer or the Macro Instruction Unit are decoded into requests for computational resources. The resource manager then allocates the required resources by consulting a table of busy resources.

Execute — The instructions are issued for execution to the three-stage integer execution unit or the floating-point execution unit, each with its own set of registers and its own ALU. Up to four instructions can be executing at the same time—three integer instructions and one floating-point calculation.

Figure 5.2 shows the three phases of the CLIPPER pipeline. The fetch and decode phases constitute the *upper pipeline*; the parallel execution units of the last phase constitute the *lower pipeline*. Since the CLIPPER arithmetic and logical instructions operate only on registers, there is

Figure 5.2 CLIPPER Pipeline



no need for a special "address pipeline" or separate effective address calculation phases. Those instructions that require an effective address calculation simply make one pass through the integer execution unit's ALU, since these instructions will not require the ALU for arithmetic or logical operations.

Notice that the figure shows a cache prefetch subphase to the fetch phase. This item refers to the instruction CAMMU's prefetch mechanism. The I-CAMMU maintains a copy of the Instruction Pointer and fetches lines of code into the cache ahead of the fetch from the CPU. This prefetch helps the CPU to keep the instruction buffer full and improves the cache hit rate for the I-CAMMU.

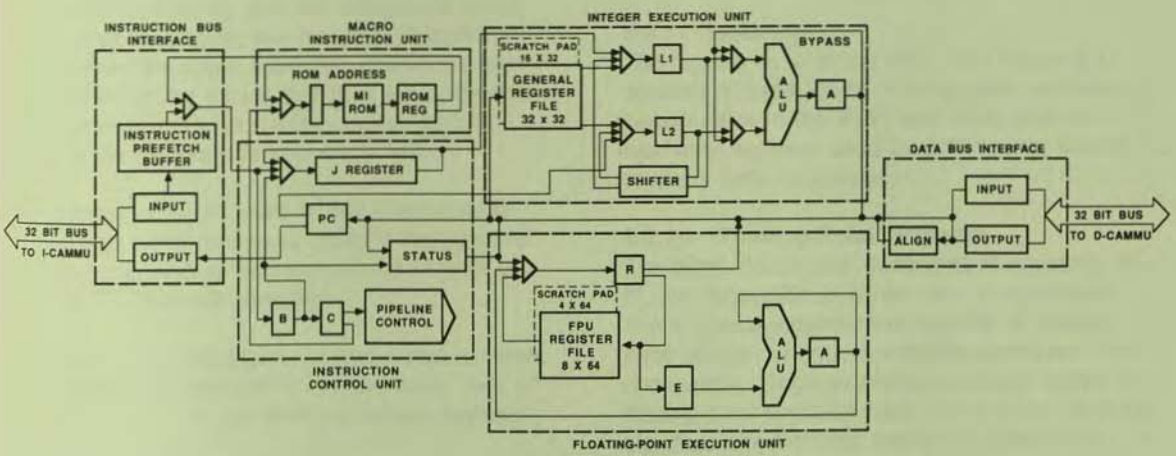
The resource manager keeps a table of all resources and which instruction is using each one; it also maintains the status of all instructions that are currently executing. Because of this detailed tracking, the CLIPPER can restart instructions that cause page faults or continue instructions after interrupts and traps. Programs, interrupts, and traps cannot crash the pipeline.

5.3 DUAL EXECUTION UNITS

The CPU contains two execution units that comprise the final phase of the pipeline. The integer execution unit handles all address calculations and all integer arithmetic operations except multiply, divide and modulus. The floating-point execution unit handles floating-point arithmetic and the integer multiply, divide and modulus operations.

The integer execution unit is itself subdivided into three stages, each of which can proceed concurrently and can be overlapped for successive instructions. The three are called the L stage, the A stage, and the O stage. Figure 5.3 shows a detailed block diagram of the CPU, which can be useful in tracing what happens in each stage.

Figure 5.3 CLIPPER CPU Block Diagram



L (Load) stage	In this first stage, operands are read from the general register file into the L registers. Immediate operands are taken directly from the instruction buffer to the L registers, via the J register.
A (Accumulate) stage	In this second stage, arithmetic, logical, or shift operations are performed on the operands in the L registers or on intermediate results from the last operation, and the output is stored in the A register.
O (output) stage	In this final stage, the contents of the A register are sent to the FPU, stored in the general register file, fed back via the bypass loop to the A stage (intermediate results), or send out to the data CAMMU.

The bypass loop makes it possible for the intermediate results of multi-instruction calculations to be fed back immediately to the next instruction.

The instructions from the Macro Instruction Unit make use of their own register files, so they do not require general register resources being used by ordinary instructions. In addition, they can use all the other registers on the CPU.

The Floating-Point Unit performs single- and double-precision floating-point operations concurrently with the integer execution unit, using its own ALU and set of eight, 64-bit registers. Because the FPU is on the CPU chip, floating-point operations do not require any additional bus transfers; thus bus traffic is reduced and performance is improved. All CLIPPER floating-point arithmetic operations conform to the IEEE 754 standard.

Chapter 6
Floating-Point Unit

6.1 INTRODUCTION

The CLIPPER CPU chip contains an integrated Floating-Point Unit (FPU). Because of this high level of integration, floating-point operations can employ the same cacheing, pipelining, and memory management mechanisms as integer operations. The result is faster floating-point execution (greater than one million single-precision floating-point operations per second). All CLIPPER floating-point operations conform to the IEEE 754 standard.[1]

This chapter describes the CLIPPER FPU from the user's point of view. The first section explains the basic programming model of the FPU and the steps required to initialize it. An overview of the requirements of the IEEE Standard is provided, and subsequent sections describe how the CLIPPER FPU implements those requirements. In the remaining sections, the arithmetic operations of the hardware, along with the exceptional conditions that may arise, are covered. Operations other than the basic arithmetic operations are described to the extent that they are supported in the hardware. The software required to fully utilize the hardware is also explained.

A number of terms used in this chapter have meanings which are more specific than general usage, and whose meanings are mandated by the IEEE 754-1985 standard:

biased exponent — the exponent plus a bias constant which is $2^7 - 1$ for single- and $2^{10} - 1$ for double-precision formats

exponent — a power of two which when multiplied by the significand gives the magnitude of a floating-point number

sign — a bit which when true indicates a negative value and when false indicates a positive value

significand — a string of bits which for finite normalized floating-point numbers consists of an explicit or implicit leading bit to the left of its binary radix point and a fraction field to the right

ulp — unit in the least significant place of the significand

1. IEEE 754-1985, *Standard for Binary Floating-Point Arithmetic.*

6.2 PROGRAMMING MODEL

This section describes the basic FPU programming model, including registers, data types and instructions, and the initialization of the FPU to the IEEE Standard environment.

6.2.1 Registers

The register map of the FPU (see Figure 6.1) consists of eight, 64-bit floating-point registers, certain fields in the PSW and SSW, and two read-only registers used by software for correct recovery from exceptions.

6.2.1.1 Floating-Point Registers

The eight, 64-bit user floating-point registers, f0–f7, are accessible in either user or supervisor mode. These registers are capable of holding either single- or double-precision operands interchangeably. Single-precision operands stored in floating-point registers have zeros in the 29 least significant fraction bits and in the three most significant exponent bits.

All of the floating-point instructions are register-to-register operations. The floating-point registers may be directly loaded from memory or may be loaded by transfer from the integer register file.

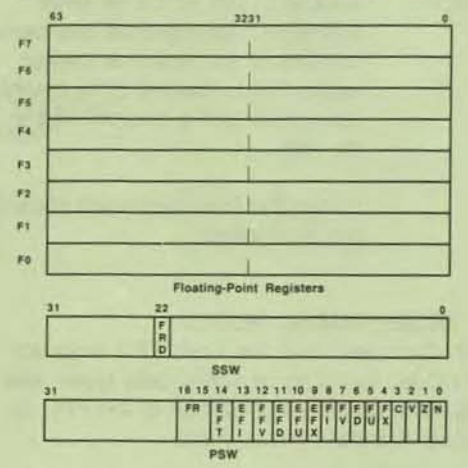
Storing of operands may be direct to memory or by transfer to the integer register file.

The integer multiplication, division and modulus instructions, though executed by the FPU, use the integer register file (r0–r16).

6.2.1.2 PSW

The PSW fields of importance to the operation of the FPU are shown in Figure 6.1.

Figure 6.1 FPU Programming Model



The *C, V, Z and N condition codes* are used for determining the destination of conditional branch instructons. The FPU sets the state of the V condition code at the end of integer multiplies, divides and mods, and sets the state of the Z and N condition codes at the end of floating-point compare instructions.

The *floating-point exception flags* (the F* group) are the so-called "sticky flags" for the five exceptions. They are called "sticky" because they are set by hardware (when the exception occurs), but are never cleared by the hardware; they may be set, cleared, saved or restored by user software. The floating-point exception flags may be cleared by creating a negative mask, ANDing it with the PSW, and writing the result to the PSW. For example,

```
movpw   psw, r0
andi    $0xFFFFFE0F, r0
movwp   r0, psw
```

clears all of the floating-point exception flags. The behavior of these flags is described in Section 6.9.2, Floating-Point Exceptions.

The *enable floating flags* (the EF* group) are the trap enable flags. The sense of these is that when EFT is false, no trap is taken. If EFT is true and the individual trap enable flag is also true at the time of a specific floating-point exception, then a trap is taken. See Section 6.9 for details.

The *floating rounding mode flags* (FR1 and FR0) select one of the four rounding modes:

FR1	FR0	Rounding Mode
0	0	Round nearest
0	1	Round toward + infinity
1	0	Round toward − infinity
1	1	Round toward zero

The default rounding mode is *round to nearest*, also known as unbiased rounding. Unbiased rounding means that the rounded result is the closest representable number to the exact result, with an exact tie broken in favor of the representable number with zero as its least significant bit. In this way, exact ties tend to statistically round up or round down an equal number of times.

Round toward plus infinity and *round toward minus infinity* are the directed roundings. Round toward plus infinity gives as a result the nearest representable number which has a greater value than (lies to the right of on a number line) the infinitely precise result. Round toward minus infinity yields as a result the nearest representable number which is less than (lies to the left of on the number line) the infinitely precise result. The directed roundings are provided to enable carefully constructed programs to implement interval arithmetic.

Round toward zero, also known as truncate, simply chooses for the result the nearest number with a smaller magnitude than the infinitely precise result. This is readily accomplished by simply chopping off all bits to the right of the last one which is representable. Round toward zero is

inherently the fastest rounding mode to implement.

6.2.1.3 SSW

The SSW's *floating register dirty flag* is provided to reduce context swtiching overhead for floating-point operations. This flag is set by hardware whenever a floating-point register is written; it can subsequently be cleared by the operating system to indicate that no floating-point register has been written and that a save of the registers is not required.

6.2.1.4 Floating Exception Registers

There are two read-only registers, which contain the PC pointing to the last floating-point instruction to have been executed and the value formerly in the destination register of that instruction. These registers are read using the instruction

```
loadfs      w1, d2
```

which transfers the PC of the last floating-point instruction executed to w1. And it transfers the value of the destination register of the floating-point instruction immediately preceding the executed floating-point instruction to floating register d2.

Using **loadfs** as the first instruction in a trap handler, the state of the FPU immediately before the exception occurred is recreated, and the instruction which caused the exception is retrieved using the address in w1.

loadfs is only defined for use by a trap handler after a floating-point exception. The use of **loadfs** in any place other than a trap handler which has interrupts disabled may give incorrect results.

6.2.2 Instructions and Data Types

The FPU performs the floating-point addition, subtraction, multiplication and division operations conforming to the IEEE standard 754 for floating-point arithmetic in the single- and double-precision formats. It also performs integer multiplication, division and modulus operations.

Floating-point comparison operations are provided for both precisions. Results of the comparison are indicated in the PSW condition codes at the conclusion of a floating-point comparison, and may be used by the conditional branch instructions.

Macro sequences provide negation, scaling and conversions for both precisions.

6.2.3 Initialization

The first step in initializing the FPU is handled automatically by hardware reset (see Section 7.2.9, Reset and Diagnostics). Resetting the CLIPPER eliminates potential inconsistencies in the logical state of the machine, which will have arisen due to random values contained in the control flip-flops. The Standard requires that the initial condition of the mode controls of any floating-point unit be as follows:

1. The rounding mode must be round nearest.
2. All exception flags must be cleared.
3. All traps must be disabled, enabling default results for all exceptions.

Following reset, the behavior of the machine is under software control. The software initialization of the FPU, described below, should be performed by the operating system every time a new process is started.

In the interest of providing a secure environment, the contents of each data register should be set to a known value. Some programming languages require that these values be initialized to zero. It is more in keeping with the intent of the authors of the IEEE Standard, though not required, to use a special reserved value called a NaN (Not_a_Number). Whichever of these is chosen, the value should be created and then loaded into each of the floating-point registers.

The following code initializes the FPU:

```
loadq   $0, r0          # Create zero.
movwp   r0, psw         # Load PSW with rounding
                        # mode 00 (nearest) and clear
                        # all exception flags and trap
                        # enable bits.
```

```
movld   r0, f2          # store in floating register 2
movld   r0, f3          # store in floating register 3
movld   r0, f4          # store in floating register 4
movld   r0, f5          # store in floating register 5
movld   r0, f6          # store in floating register 6
movld   r0, f7          # store in floating register 7
loadi   $0x7ffbad75,    # high half of hex NaN
        r1              # 1.bad75a
loadi   $0x00000000,    # low half of NaN
        r0
movld   r0, f0          # store in floating register 0
movld   r0, f1          # store in floating register 1
```

The NaN used in the initialization above is known as a quiet NaN. The properties of NaNs will be discussed further in Sections 6.5 and 6.6, but for now it is useful to observe that a quiet NaN propagates through arithmetic operations unchanged (except for possibly the sign bit), so that any user who operates on a register not yet defined will receive a NaN as a result of the operation.

6.3 DATA MOVEMENT OPERATIONS

This section describes the various ways to move data within the FPU, and from the FPU to the CPU and memory.

There are four ways to enter data into the floating-point registers. The first two move data from general registers to the FPU; the second pair move data from memory to the FPU:

```
movws   r3, f2      # Move general register 3 contents
                    # to floating register 2, treating the
                    # former as having single-precision
                    # type.
movld   r2, f1      # Move contents of general register
                    # pair r2,r3 into floating register 1,
                    # treating the former as having
                    # double-precision type.
loads   sgl, f5     # Find the label sgl. At this address
                    # in memory read the contents and
                    # transfer them both to floating
                    # register 5, treating the value as
                    # having single-precision type.
loadd   dbl, f0     # Find the label dbl. From this
                    # address and the next highest
                    # word address in memory read the
                    # contents and transfer them both
                    # to floating register 0, treating the
                    # value as having double-precision
                    # type.
```

There are four ways to move data out of the FPU. The first pair move data from the FPU to general registers, and the second pair move the data to memory:

```
movsw   f2, r3      # Move contents of floating regis-
                    # ter 2, regarded as being in the
                    # single format, to general regis-
                    # ter 3.
movdl   f1, r2      # Move contents of floating regis-
                    # ter f1, regarded as being in the
                    # double format, to general regis-
                    # ter pair r2,r3. The sign,
                    # exponent and most significant
                    # fraction bits are in r3, the least
                    # significant fraction bits are in
                    # r2.
stors   f5, sgl     # Find the label sgl. Write the
                    # contents of floating register 5,
                    # formatted in the single-precision
                    # type, into this memory location.
                    # Since CLIPPER memory is
                    # addressed by bytes, it is
                    # essential that the two lsbs of
                    # the address be zeros.
stord   f0, dbl     # Find the label dbl. Write the
                    # contents of floating register 0,
                    # formatted in the double-preci-
                    # sion type, into this memory
                    # location and the next highest
                    # word address. Since CLIPPER
                    # memory is addressed by bytes,
                    # it is essential that the three
                    # lsbs of the address be zeros.
```

There are two move instructions, one for single-precision and another for double-precision, that copy data from one floating-point register to another.

```
movs    f3, f7      # Copy the contents of floating
                    # register 3 into floating register
                    # 7 with all exponent bits more
                    # significant than the eighth bit
                    # set to zero, and all fraction bits
                    # less significant than the twenty-
                    # third bit also set to zero.
```

```
movd    f3, f7  # Copy the contents of floating
                # register 3 into floating register
                # 7 without modifying any of the
                # bits.
```

It is useful to observe here that if a single-precision variable is moved by the **movd** instruction, it is not modified. The reverse situation does not hold: a double-precision variable move by the **movs** instruction has its fractional part truncated after the twenty-third bit, and its exponent stripped of its 3 most significant bits. Also, **stord** executed on a single-precision operand results in an inaccurate value if the memory value is interpreted as if it were double-precision; but if the memory value is restored by a **loadd** instruction, the original single-precision value is restored when viewed as a single-precision value. This is very useful for context switching, because it is feasible to save and restore any floating-point register using **loadd** and **stord**, without regard for the presence of single-precision operands in the FPU registers.

6.4 INTEGER OPERATIONS

The CLIPPER FPU performs four basic operations which support the CLIPPER's eight integer multiply and divide instructions. These operations may result in the two integer exceptions, integer overflow and integer divide by zero. Integer overflow is possible on signed and unsigned multiplies, as well as on signed divides and mods. Integer divide by zero is possible on signed divides and mods.

6.4.1 Integer Multiply

The two multiplies are signed integer multiply and unsigned integer multiply. Signed integers range from $+2^{31} - 1$ (hex 7FFF FFFF) down to zero (hex 0000 0000) to minus one (hex FFFF FFFF) down to -2^{31} (hex 8000 0000). Signed integer multiply supports the instructions **mulw** and **mulwx**: For example:

```
mulw    r3, r9  # Multiply the contents of general
                # register r3 by the contents of
                # general register 9, treating both
                # as signed two's complement inte-
                # gers. Write into register 9 the 32
                # least significant bits of the prod-
                # uct. If the 32 msbs of the prod-
                # uct are not just the sign
                # extension of the 32 lsbs, then
                # signal integer overflow.
```

For Example:

```
loadi   $0x55555, r3
loadi   $0xFF800008, r1
mulw    r3, r1
```

This code I oads the number 349525 (hex 55555) into general register 3, loads the number −8388600 (hex FF800008) into general register 1, and then multiplies the two signed integers. The result is FFFFFD55 55AAAAA8 or hex −2aaaa555558. The hex value 55AAAAA8 is returned to r1, and integer overflow is signalled because the result did not fit into the least significant word. If this example is rerun using the extended instruction, the result is different:

```
loadi   $0x55555, r3
loadi   $0xFF800008, r4
mulwx   r3, r4
```

The product is again FFFFFD55 55AAAAA8, but now 55AAAAA8 is delivered to general register r4, and FFFFFD55 is delivered to r5, considered as the upper half of the longword r4. Again, since the result does not fit in one word, the integer overflow is signalled.

Unsigned integer multiplies behave similarly, but both operands are always positive, and the magnitudes represented range from zero (hex 0000 0000) up to $2^{32} - 1$ (hex FFFF FFFF):

```
mulw    r3, r16 # Multiply the contents of
                # general register r3 by the
                # contents of general regis-
                # ter 6, treating both as
                # unsigned integers. Write
                # into register 6 the 32
                # least significant bits of the
                # product, and write the
                # 32 msbs of the product
                # into register 7, considered
                # as the upper word of
                # register pair l6. If the
                # 32 msbs of the product
                # are not exactly zero then
                # signal integer overflow.
```

Rerunning the above example:

```
loadi   $0x55555, r3
loadi   $0xFF800008, r1
mulwu   r3, r1
```

In this case, the value loaded into r3 is 349525 (hex 55555). The value loaded into r1 is again hex FF80 0008, but since it is now an unsigned integer, the value is 4286578696. The unsigned product is 000552aa 55aaaaa8. The product's lsw (55aaaaa8) is delivered to general register r1, and integer overflow is signalled.

Finally, if the instruction is **mulwux** (multiply word unsigned extended):

```
loadi   $0x55555, r3
loadi   $0xFF800008, r4
mulwux  r3, r4
```

In the above example, 55aaaaa8 is loaded into r4, and 000552aa is loaded into register r5, which is considered the msw of register pair r4. Overflow is signalled.

The extended multiplies are available to support extended precision integer arithmetic. The non-extended multiplies are generally used outside of this application.

6.4.2 Integer Divide and Mod

There are four integer divide instructions: **divw**, **divwu**, **modw** and **modwu**. All four take a pair of values, a and b, and deliver q or r such that the following rules are always satisfied:

1. $b = q \times a + r$
2. sign (r) = sign (b)
3. $0 \leq |r| < |a|$

Notice that the order of the operands is important: the first named operand is divided into the second named operand. For example,

```
loadi   $37,  r2
loadi   $4,   r1
divw    r1,   r2
```

computes the quotient q = 9 and the remainder r = 1. Since the instruction is **divw**, the quotient 9 is written into r2. If the **mod** instruction is used, for example,

```
loadi   $37,  r2
loadi   $4,   r1
modw    r1,   r2
```

then the quotient is still 9 and the remainder is still 1. In this case the remainder 1 is written into r2.

For signed division, there is a special case which results in integer overflow:

```
loadi   $-2147483648,  # hex 8000 0000
        r2
loadi   $-1, r1        # hex FFFF FFFF
divw    r1, r2
```

The result is 2147483648, which cannot be properly expressed as a signed 32-bit integer. This is the only case which results in an integer divide overflow; these same operands will also signal an integer overflow (by setting the V condition code flag in the PSW) for signed modulus.

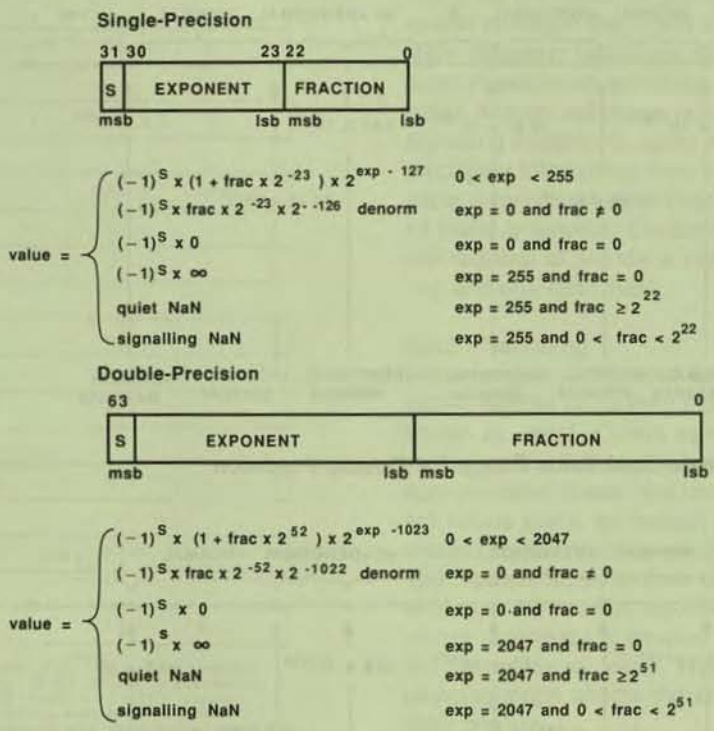
Unsigned integer divides and mods behave similarly. The quotient is always positive or zero, and the remainder is also always positive or zero. Unsigned integer divides and mods never result in integer overflow.

For both signed and unsigned integer divides and mods, if the divisor (the A operand) is zero, then the result is undefined and the integer divide by zero exception is signalled.

6.5 OVERVIEW OF THE IEEE STANDARD 754 FOR BINARY FLOATING-POINT ARITHMETIC

The CLIPPER FPU was designed to support the IEEE Standard 754 for Binary Floating-Point Arithmetic. Familarity with ANSI/IEEE 754-1985 (hereafter referred to simply as the Standard), though not required to understand CLIPPER, is a valuable first step for those unfamilar with float-

Figure 6.2 Floating-Point Formats



ing-point arithmetic implementations. A copy of the Standard can be obtained directly from IEEE Service Center, Publication Sales Dept., 445 Hoes Lane, Piscataway, N.J. 08854; (201) 981-0060.

The IEEE committee responsible for drafting Standard 754 was made up of a broad base of computer manufacturers, users, and academicians. Their main goal was to standardize methods for implementing floating-point arithmetic in order to allow portability of numerically precise application programs at a reasonable cost. Prior to the development of the Standard, diverse (i.e., incompatible) implementations abounded. The results of their labor have been fruitful: virtually all new implementations, including CLIPPER, are based on the Standard.

The Standard specifies floating-point formats, rounding, supported operations, methods for dealing with out of range and undefined values, and a uniform approach to exceptions and traps. It is important to note that the Standard does not define the method of implementation, but only the programming environment as a whole. Thus any combination of hardware and software may be utilized in a particular implementation.

6.5.1 Formats

The Standard defines four floating-point formats: single, single extended, double, and double extended. From an applications standpoint, the tradeoffs between precisions concern the required range and precision of the particular application, the performance needed, and the storage requirements. The most commonly implemented formats

Figure 6.3 Approximate Real Value Range for Single-Precision

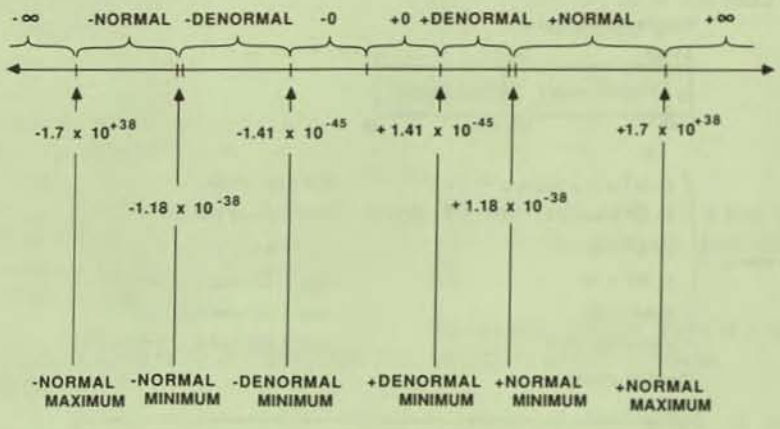
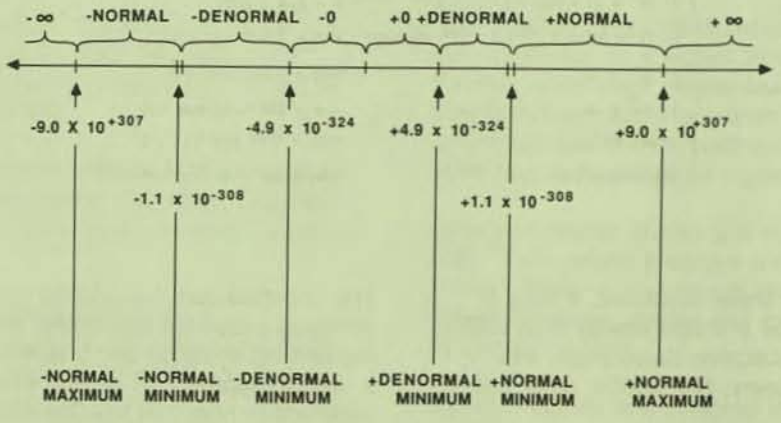


Figure 6.4 Approximate Real Value Range for Double-Precision

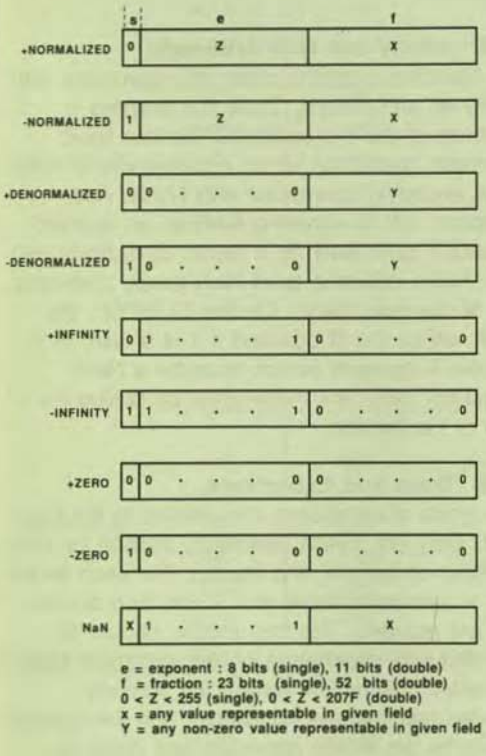


are single- and double-precision, represented by 32 and 64 bits respectively (see Figure 6.2). Single- and double-precision will be discussed in the remainder of this section. Refer to the Standard for specifics on extended formats. Binary floating-point numbers can be represented by the general formula $(-1)^s \times 2^E \times f$, where s is a sign bit, E an exponent, and f a significand. The Standard defines an exponent offset, or bias, to be added to the nonbiased exponent, E. A bias is used to force all exponent values non-negative, which ensures certain useful mathematical properties. The bias value is 127 for single-precision and 1023 for double-precision. The nomenclature assigns e as the biased exponent, where e = E + bias.

6.5.2 Encoding of Values

The length of the single and double bit fields allow a maximum of either 2^{32} or 2^{64} unique values to be represented. The Standard apportions these representations assignments among

Figure 6.5 IEEE 754 Bit Encodings for Single- and Double-Precision Values



the finite representable real numbers (normalized, denormalized and ±zero) representations for out of range values (±infinity), and a class of entities known as NaNs, which are explained below. Figures 6.3 and 6.4 show the range and precision available for both single and double formats. Single-precision is able to represent numbers between approximately 10^{-45} to 10^{38}. Double-precision extends the range and precision to approximately 10^{-324} to 10^{307}.

Values are also reserved for the case where the biased exponent is equal to it's maximum value (i.e., all 1's). This occurs for single-precision when e=255 and for double-precision when e=2047. If the fractional portion is zero, +infinity and −infinity are represented. A nonzero fractional portion represents a NaN (Not a Number). Figure 6.5 summarizes the bit encodings as defined by the Standard.

NaNs are reserved values used to encode non-numeric data. There are two types of NaNs: signalling and quiet. A signalling NaN forces an invalid operation trap, then behaves like a quiet NaN. Signaling NaNs are commonly used to support enhanced arithmetic. A quiet NaN propagates through arithmetic operations without signalling exceptions. Quiet NaNs are used to propagate information from invalid operations to allow diagnosis of error conditions. The sign bit of all NaNs is ignored. Conforming implementations are required to provide a minimum of one signalling and one quiet NaN.

6.5.3 Rounding

Within a given finite word length, some real values can be accurately represented (these are known as *exact*). Others cannot be exactly represented in this finite word length and thus require approximation (these are known as *inexact*). Inexact values signal an inexact exception when created as the result of an arithmetic operation. The representation chosen for any inexact value is always the nearest representable number whose magnitude is smaller than the value, or the nearest larger value. Thus the choice of rounding mode affects the chosen result by at most one ulp.

The method of approximation of inexact values is specified by the four rounding modes:

Round toward nearest
Round toward +infinity
Round toward −infinity
Round toward zero

Any of these modes must be selectable by the user. Round toward nearest is the default. In the case of a value equidistant from two representable values with round toward nearest, rounding is to the value with the lsb=0.

There are two representations for zero, namely +zero and −zero. The representation for exact zero depends upon the size of the operands whose result was this zero; for zero sums and differences the representation also depends on the rounding mode. For zero sums and differ-

ences where the addends were not both zero and the rounding mode is round toward nearest, zero, or +infinity, exact zero is represented by +zero. For round toward −infinity, exact zero is −zero.

6.5.4 Operations

Operations which must be supported to conform to the Standard are shown in Table 6.1, and include basic arithmetic operations, conversions, and comparison. Hardware components requiring software to comply with the Standard may not be said to be conforming without the software. Transcendental functions such as sine and cosine are not included within the scope of the Standard.

TABLE 6.1 IEEE Required Operations

Addition
Subtraction
Multiplication
Division
Square root
Remainder
Round to integer
Format conversion among each supported format
Floating-point/integer conversions
Binary/decimal conversion
Compare

The basic arithmetic operations must be supported for operands of the same format. It is recommended, though not required, that these operations also be able to handle operands of differing formats. It must be possible to convert floating-point numbers between all supported formats, as well as between floating-point and all supported integer formats. Conversion between decimal (ASCII) strings in at least one format and all supported floating-point basic formats is required.

The compare operation must be provided for all supported formats as well as operands of differing formats. The compare operation can result in one of four conditions; less than, equal to, greater than, or unordered. The unordered condition results when one or both of the operands is a NaN. The sign of zero is ignored in compare operations.

6.5.5 Infinity and NaN Arithmetic

The Standard supports arithmetic operations with infinity as an operand. Table 6.2 provides a summary of the the results of the four basic arithmetic operations for all combinations of operands, excluding operations with NaNs. Any operation with a signalling NaN as an operand delivers a quiet NaN as a result. Operations with quiet NaNs deliver a quiet NaN result, preferably one of the input NaNs. On the CLIPPER, the result will be the B operand if it is a Nan, or if not, the A operand (which must be a NaN). Signalling NaNs are made quiet by setting the msb of the fraction.

6.5.6 Traps and Exceptions

Five types of exceptions are defined in the Standard. They are invalid operation, division by zero, overflow, underflow, and inexact. For each exception, a user-controllable status and trap enable flag are required. The trap routine should be provided with information on the exception type, operation, destination format, the correctly rounded result where applicable, and the operand values for the invalid operation and divide by zero traps. The default mode is that exceptions do not cause traps and deliver Standard specified values to the destination.

6.6 FLOATING-POINT ARITHMETIC OPERATIONS

This section describes the details of floating-point arithmetic, including how various real numbers are (approximately) represented in the floating-point formats, and the operations of addition, subtraction, comparison, multiplication and division. Although the exceptions are briefly introduced, full discussion of exceptions is deferred to Section 6.9.

6.1.1 Floating-Point Formats

The CLIPPER FPU supports the Standard's single- and double-precision data types, including all special values thereof.

TABLE 6.2 Infinity Arithmetic

		Operand A					
	A+B	−inf	−finite	−0	+0	+finite	+inf
Operand B	−inf	−inf	−inf	−inf	−inf	−inf	NaN
	−finite	−inf					+inf
	−0	−inf					+inf
	+0	−inf					+inf
	+finite	−inf					+inf
	+inf	NaN	+inf	+inf	+inf	+inf	+inf

		Operand A					
	B−A	−inf	−finite	−0	+0	+finite	+inf
Operand B	−inf	NaN	−inf	−inf	−inf	−inf	−inf
	−finite	+inf					−inf
	−0	+inf					−inf
	+0	+inf					−inf
	+finite	+inf					−inf
	+inf	+inf	+inf	+inf	+inf	+inf	NaN

		Operand A					
	A×B	−inf	−finite	−0	+0	+finite	+inf
Operand B	−inf	+inf	+inf	NaN	NaN	−inf	−inf
	−finite	+inf					−inf
	−0	NaN					NaN
	+0	NaN					NaN
	+finite	−inf					+inf
	+inf	−inf	−inf	NaN	NaN	+inf	+inf

		Operand A					
	A÷B	−inf	−finite	−0	+0	+finite	+inf
Operand B	−inf	NaN	+inf	+inf	−inf	−inf	NaN
	−finite	+0					−0
	−0	+0					−0
	+0	−0					+0
	+finite	−0					+0
	+inf	NaN	−inf	−inf	+inf	+inf	NaN

6.6.1.1 Single-Precision Format

The single-precision format can approximately represent any real number between $1.70 \times 10^{+38}$ and 1.41×10^{-45}, zero, and the additive inverses of all of these. For magnitudes between $1.70 \times 10^{+38}$ and 1.18×10^{-38}, the precision of the representation is better than one part in 10^{7}. For the smallest representable numbers and denormal numbers, the accuracy is an absolute $\pm 1.41 \times 10^{-45}$. The single format also provides codes for plus and minus infinity, some signalling NaNs and some quiet NaNs.

The single-precision format, shown in Figure 6.2, consists of three fields.

The first field, located in bit position 31, is the sign bit. A zero in this field indicates a positive value, and a 1 indicates a negative value. The sign bit is not interpreted for any NaN.

The second field is the exponent field. This field is eight bits long and is located in bit positions 30..24 inclusive. The values 255 and 0 in this field have special meanings. Otherwise, the number in this field is called the biased exponent. The true exponent value is the biased exponent minus 127.

The third field is the significand field. This field is twenty three bits long and is located in bit positions 22..00. If the exponent is not 255 nor 0, then the fraction of the value is one plus the result of dividing the significand field (viewed as an unsigned integer) by 2^{23}.

The value of a single-precision floating-point number is determined by selecting the appropriate rule below. All fields are extracted as if they were unsigned binary integers:

1. If exponent = 255 and $0 <$ significand $< 2^{22}$
   then v(fp1) = NaN_signalling (signalling Not_a_Number)

2. If exponent = 255 and significand $\geq 2^{22}$
   then v(fp1) = NaN_quiet (non-signalling Not_a_Number)

3. If exponent = 255 and significand = 0
   then v(fp1) = $(-1)^{sign} \times$ (infinity) (infinities)

4. If $0 <$ exponent < 255
   then v(fp1) = $(-1)^{sign} \times 2^{-127} \times (1 + \text{significand} \times 2^{-23})$

5. If exponent = 0 and significand <> 0
   then v(fp1) = $(-1)^{sign} \times 2^{-126} \times (\text{significand} \times 2^{-23})$ (denormal number)

6. If exponent = 0 and significand = 0
   then v(fp1) = $(-1) \times 0$ (zeros)

For example, the representation in single-precision format for -10 is developed as follows:

1. Represent the number in radix two scientific notation:
   $-10 = (-1) \times 2^{(3)} \times 1.010$
   $= (-1) \times 2^{(00000011)} \times 1.01000000000000000000000$

2. Bias the exponent:
   biased exponent = exponent + 127
   = 00000011 + 01111111
   = 10000010

3. Remove implicit leading msb of the fraction to get:
   significand = 01000000000000000000000

4. Concatenate the fields:
   sign : biased exponent : significand
   1 : 10000010 : 01000000000000000000000
   Q = 11000001001000000000000000000000

As another example, represent 3.172×10^{-42} in single-precision floating-point format:

1. Represent the number in radix two scientific notation:
   $3.172 \times 10^{-42} = (+1) \times 2^{(-138)} \times 1.105280682$
   $= (+1) \times 2^{(-10001010)} \times 1.00011010111100111010110$

2. Bias the exponent:
   $= -10001010 + 01111111$
   $= -00001011$

   Since -00001011 is not directly representable, it is necessary to create a denormal representation:

   2a. Let n = $-1 \times$(biased exponent) + 1
   = 00001100 = 12(ten)

   2b. Set the biased exponent to 00000000

   2c. Right shift the fraction n places and round:

   1.00011010111100111010110
   0.00000000000100011010111100111010110
   0.00000000000100011011000

3. significand = 00000000000100011011000

4. Concatenate:

sign	:	biased exponent	:	significand
0	:	00000000	:	00000000000100011011000

Q = 00000000000000000000100011011000

As another example, identify the value of the single-precision number 10010100001000010001001011101111:

1. Segment into fields:

sign = 1
biased exponent = 00101000
significand = 01000010001001011101111

2. Determine exponent (not denormal):

exponent = biased exponent − bias
= 00101000 − 01111111
= −01010111 = −87(ten)

3. Create fraction from significand:

significand = 01000010001001011101111
fraction = 1.01000010001001011101111
= 1.260425 (ten)

4. Express in radix ten scientific notation:

value = $(+1) \times 2^{(-87)} \times 1.260425$
= $8.145306 \times 10^{(-27)}$

The internal representation for single-precision numbers is

sign : one bit
exponent : eleven bits
significand : fifty-three bits with explicit leading bit

When a single-precision number is loaded into the FPU, the sign bit is extracted from bit position 31 of the incoming word. The exponent is formed by extracting the eight bits in positions 30..24 inclusive and right justifying these in the eleven bit internal exponent field. The three more significant bits are zeroed. The fraction field is formed by extracting the twenty-three least significant bits of the incoming word and then inserting these, left justified, into the internal fraction field. All bits to the right of these are set to zero. The fraction field is then right shifted one place. If the exponent is zero, then the msb of the fraction field is set to zero; otherwise it is set to one. (The last sentence describes the creation of the implicit leading fraction bit prescribed in the standard.) For example, the incoming word

bit	3	3 2 2 2 2 2 2 2	2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
position:	1	0 9 8 7 6 5 4 3	2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
fields:	s	exponent	significand

is represented internally as

```
sign      :=    31
exponent  :=    $0 $0 $0 30 29 28 27 26 25 24 23
fraction  :=    $1 $. 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08
                07 06 05 04 03 02 01 00 $0 $0 $0 $0 $0 $0 $0 $0
                $0 $0 $0 $0 $0 $0 $0 $0 $0 $0 $0 $0 $0 $0 $0 $0
                $0 $0 $0 $0 $0
```

where $ prefix signifies a constant, and no prefix signifies that a bit position from the incoming word is copied to this bit position.

6.6.1.2 Double-Precision Format

The double-precision format can approximately represent any real number between $9.0 \times 10^{+307}$ and 4.9×10^{-324}, zero, and the additive inverses of all of these. For magnitudes between $9.0 \times 10^{+307}$ and 1.1×10^{-308}, the precision of the representation is better than one part in 10^{16}. For the smallest representable (denormal) numbers, the accuracy is an absolute $\pm 1.1 \times 10^{-308}$. The double format also provides codes for plus and minus infinity, some signalling NaNs, and some quiet NaNs.

The double-precision format, shown in Figure 6.1, consists of three fields. The first field, located in bit position 63, is the sign bit. If this is a 0, then the value is positive. If the sign bit is a 1, then the value is negative.

The sign bit is not interpreted for any NaN.

The second field is the exponent field. This field is eleven bits long and is located in bit positions 62..52 inclusive. The values 2047 and 0 in this field have special meanings. Otherwise, the number in this field is called the biased exponent. The true exponent value is the biased exponent minus 1023.

The third field is the significand. This field is fifty-two bits long and is located in bit positions 51..00. If the exponent is neither 2047 nor 0, then the fraction of the value is one plus the result of dividing the significand by 2^{52}.

The value of a double-precision floating-point number is determined by selecting the appropriate rule below. All fields are extracted as if they were unsigned binary integers.

1. If exponent = 2047 and 0 < significand < 2^{51}
   then v(fp2) = NaN_signalling

2. If exponent = 2047 and significand > = 2^{51}
   then v(fp2) = NaN_quiet

3. If exponent = 2047 and significand = 0
   then v(fp2) = $(-1)^s \times$ (infinity)

4. If 0 < exponent 2047
   then v(fp2) = $(-1)^{sign} \times 2^{(exponent - 1023)} \times (1 + significand \times 2^{-52})$

5. If exponent = 0 and significand <> 0 (*denormalized*)
   then v(fp2) = $(-1)^{sign} \times 2^{-2046} \times (significand \times 2^{-52})$

6. If exponent = 0 and significand = 0
   then v(fp2) = $(-1)^s \times 0$

As an example, create the double-precision representation of minus infinity:

1. Infinity is a special case. Use rule 3. above.

2. Sign = 1 since value is negative.

3. Biased exponent = 2047(ten) = 11111111111(two)
   significand = 0(ten) = 0(two)

4. Concatenate fields:
   1 : 11111111111 : 000000000000...000

   11111111111100000000000000000000 hi word
   00000000000000000000000000000000 lo word

As another example, identify the double-precision value:

01111111111111001100110011001100 hi word
01010101010101010101010101010101 lo word

1. Separate the fields:
   0 : 11111111111 : 10011001100110011000101010101010101010101010101010101
2. Biased exponent is 2047, special case.
3. Significand $> 2^{51}$ so invoke rule 2.
4. This is a quiet NaN.

When a double-precision number is loaded into the FPU, the sign bit is extracted from bit position 31 of the second incoming word. The exponent is formed by extracting the eleven bits in positions 30..20 inclusive of the second incoming word, and placing these in the internal exponent field. The fraction field is formed by concatenating the remaining twenty bits of the second word to the left of all thirty-two bits of the first (less significant) word and placing this, left justified, into the internal fraction field. The fraction field is then right shifted one place. If the exponent is zero, then the msb of the fraction field is set to zero; otherwise it is set to one. For example, the incoming words

bit position:	6 3	6 2	6 1	6 0	5 9	5 8	5 7	5 6	5 5	5 4	5 3	5 2	5 1	5 0	4 9	4 8	4 7	4 6	4 5	4 4	4 3	4 2	4 1	4 0	3 9	3 8	3 7	3 6	3 5	3 4	3 3	3 2
fields:	s	exponent											significand																			

	3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0
	significand																															

are represented internally as

```
sign     :=    63
exponent :=    62 61 60 59 58 57 56 55 54 53 52
fraction :=    $1 . 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37
               36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
               20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05
               04 03 02 01 00
```

Any double-precision operation delivers an 11-bit exponent, a 1-bit sign and a 53-bit fraction result. A single-precision operation delivers an 8-bit exponent with the upper three bits of the internal representation set to zeros, and a 24-bit fraction left justified with the remaining 29 lsbs set to zero. For this reason, it is safe to perform **movd** on a single-precision value, since it will not be changed; but it is ill advised to use the **movs** instruction with a double-precision value, since this will eliminate three leading exponent bits and truncate the fraction at 24 bits.

6.6.1.3 Not A Number

The Standard requires the availability of at least one signalling NaN and at least one quiet NaN. NaNs are defined in the standard as follows:

> "NaN. Not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs. Signalling NaNs signal the invalid operation exception whenever they appear as operands. Quiet NaNs propogate through almost every arithmetic operation without signalling exceptions."

The Standard refers to NaNs again:

> "Signalling NaNs afford values for uninitialized variables and arithmetic-like enhancements (such as complex-affine infinities or extremely wide range) that are not the subject of the standard. Quiet NaNs should, by means left to the implementor's discretion, afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the NaNs be preserved through arithmetic operations and floating-point format conversions."

In other sections the Standard requires that the result of any operation in which a NaN appears as an operand must be a quiet NaN. When one of the operands is a a NaN, that same NaN should be returned as the result; when both operands are NaNs, the result should be one of those NaNs.

The use of signs for NaNs is described in Section 6.3 of the Standard:

> "This standard does not interpret the sign of a NaN. Otherwise the sign of a product or quotient is the Exclusive Or of the operands' signs; and the sign of a sum differs from at most one of the addends' signs."

Taking all of the above requirements into account, the CLIPPER FPU deals with NaNs as follows:

1. If both operands are NaNs, then the result is the NaN originally found in the destination register.
2. If only one operand is a NaN, then the result is that NaN.
3. If the NaN which is chosen for the result is a signalling NaN, then the result is that NaN made quiet. This is accomplished by setting the most significant bit of the fraction (the second most significant bit of the significand) to 1.
4. The sign of a NaN is not preserved through arithmetic operations. In particular, on multiplies and divides with the other operand negative, the sign will be reversed.
5. When an operation gives an invalid result and neither operand was a NaN, the hardware provides a default quiet NaN. The default quiet NaN in single is
   s11111111111111111111111111111111
   where s is the sign bit and does not matter. The default quiet NaN in double is
   s11111111111111111111111111111111111111111111111111111111111111.

6.6.1.4 Infinities

The standard requires that plus and minus infinity be provided. The representation of infinity consists of a sign bit (true ⩾ negative sign) in bit position 31 or 63 (single or double respectively) followed by an exponent field of all ones in bit positions 30..23 or 62..52, followed by zeros in bit positions 22..00 or 51..00.

Infinities behave as one would expect for numbers at the limit for very large numbers. The behavior of infinities in addition, subtraction, multiplication and division operations is shown in Tables 6.2 through 6.5.

It is worth noticing that subtracting a signed infinity is like adding an opposite signed infinity. Also useful is the observation that dividing by a signed zero is very much like multiplying by the same signed infinity, and that dividing by a signed infinity is like multiplying by the same signed zero, especially since the NaN singularities then coincide.

6.6.2 Addition and Subtraction

Subtraction is always viewed as adding the complement, i.e., b − a is viewed as b + (− a).

Addition of floating-point numbers is more complex than addition of integers. One reason is that the exponents usually differ between the two operands. In order for the addition to be done with proper regard for the place values of the digits, it is necessary to right shift the operand having the smaller exponent by the number of places equal to the difference between the two operands' exponents. This is called the alignment step. Another difficulty arises from the representation of the fractions in sign and magnitude form. Addition of numbers with opposite signs (or subtraction of numbers with like signs) is accomplished by complementing the fraction of one operand and selecting for the preliminary sign of the result the sign of the operand not complemented. The complemented fraction is added to the fraction of the other operand. If the sum is negative, then one must complement the resulting fraction and complement the preliminary sign of the result as well.

Finally, the sum must be stated in a representable form. Since the sum of the fractions may have carried into the next more significant place, it may be necessary to left shift the result fraction by one place and update the exponent of the result by adding one to it. Alternately, the magnitude of the two values may be very similar but the signs are effectively opposite. This leads to an effect called cancellation, in which event the calculated sum will have some leading zeros. The number of leading zeros is counted; let us call this n. The fraction result is shifted left by n places. Then n is subtracted from the exponent. This process is called *normalization*. After normalization, the sum may have more significant bits than the fit in the destination format. In this case the result is inexact. Inexact results are approximated by representable numbers through the process of rounding.

Floating-point addition is subject to four possible exceptions: floating invalid operation, inexact, underflow and overflow.

Here is an example of a single-precision addition:

```
loadi     $0xbf8ccccd, r0
movws     r0, f0
loadi     $0xc27ffee1, r0
movws     r0, f1
adds      f0, f1
stors     f1, place
place:    .single
```

After the first **loadi** and **movws**, the contents of f0 is:

sign	:=	1
exponent	:=	00001111111
fraction	:=	1.00011001100110011001101000000000000000000000000000000

After the second **loadi** and **movws**, the contents of f1 is

sign	:=	1
exponent	:=	00010000100
fraction	:=	1.11111111111111011100001000000000000000000000000000000

The difference of exponents is 00010000100 − 000001111111 = 101, so the first operand must be shifted right five places; the preliminary exponent is 00010000100 and the preliminary sign is 1. After the fraction from f0 is shifted, the sum is

frac(0)	:=	0.00001000110011001100110011010000000000000000000000000000000
frac(1)	:=	1.11111111111111011100001000000000000000000000000000000
		10.00001000110010101000111011010000000000000000000000000000000

This has carried, so the preliminary exponent becomes 00010000100 + 1 ⩾ 00010000101 and the fraction becomes

1.00000100011001010100011101101000000000000000000000000000000

This value has too many bits, so it must be rounded. Let us assume the rounding mode is round nearest (the default mode). The underlined bits must be rounded off:

1.00000100011001010100011101101000000000000000000000000000000

The round nearest rule as stated in the Standard:

> "..In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered."

The tie breaker in this rule is fair, going up half the time and down half the time. This is why this rule is also called unbiased rounding. Rounded in this way the fraction becomes

1.00000100011001010100100000000000000000000000000000000

So the result in f1 is

```
sign        :=      1
exponent    :=      00010000101
fraction    :=      1.00000100011001010100100000000000000000000000000000000
```

which is stored in memory at location "place" as

```
                    11000010100000100011001010100100
```

Here is an example of subtraction:

```
loadi       $0xcbe620ee, r0
movws       r0, f0
loadi       $0xcbe620da, r0
movws       r0, f1
subs        f0, f1
stors       f1, place2
place2:     .single
```

After the first **loadi** and **movws**, the contents of f0 is

```
sign        :=      1
exponent    :=      00010010111
fraction    :=      1.11001100010000011101110000000000000000000000000000000
```

After the second **loadi** and **movws**, the contents of f0 is

```
sign        :=      1
exponent    :=      00010010111
fraction    :=      1.11001100010000011011010000000000000000000000000000000
```

The difference of exponents is 00010010111 − 00010010111 = 0, so by convention f0 is shifted, but by zero places. Since the operands have the same sign but the operation is subtraction, the f0 fraction is also complemented. The preliminary exponent is 00010010111 and the preliminary sign is 1.

```
frac(0)     =       01.11001100010000011101110000000000000000000000000000000
```

so

```
−frac(0)    =       10.00110011101111100010001111111111111111111111111111111
                    +0.00000000000000000000000000000000000000000000000000001
```

the sum is set up:

−frac(0)	=	10.00110011101111100010001111111111111111111111111111
frac(1)	=	01.11001100010000011011010000000000000000000000000000
q	=	11.11111111111111111011000000000000000000000000000000

This is a negative result, so it is complemented and the preliminary sign is reversed to sign = 0

q	=	11.11111111111111111011000000000000000000000000000000
comp(q)	=	00.00000000000000000100111111111111111111111111111111
		+ 1
		00.00000000000000000101000000000000000000000000000000

This is not in normal form, so it must be normalized. The fraction must be shifted left by 19 places to get to normal position:

1.01000000000000000000000000000000000000000000000000

and the preliminary exponent must be decreased by 19 (10011 binary) to 00010010111 − 10011 ⩾ 00010000100. The fraction is exact, so no rounding is required. The result in f1 is

sign	:=	0
exponent	:=	00010000100
fraction	:=	1.01000000000000000000000000000000000000000000000000

and this is stored in location "place2" as

01000010001000000000000000000000.

Addition and subtraction may result in an overflow exception when two large numbers are added (at least one of them has to have a biased exponent of 254 single or 2047 double) and the fraction addition (or the subsequent rounding) has a carry out of the msb position. The overflow may coincide with the fraction sum being inexact, although this is not necessarily the case. (More details on the floating overflow exception are given in Section 6.9.) Underflow may also occur with addition or subtraction, although the result is never inexact. (Underflow is also detailed in Section 6.9.)

The IEEE Standard specifies a special set of rules for the case where the sum of two numbers b + a (or b − a regarded as b + −(a)) is exactly zero:

"When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be "+" in all rounding modes except round toward minus infinity, in which mode that sign shall be "−". However, x + x = x − (−x) retains the same sign as x even when x is zero."

These rules are scrupulously followed in the CLIPPER FPU. Some examples are:

```
 20.44 −  +20.44  =  +0.00
−20.44 −  (−20.44) = −0.00
+0.00 +   +0.00   =  +0.00
+0.00 −   (−0.00)  =  +0.00
−0.00 +   −0.00   =  −0.00
−0.00 −   +0.00   =  −0.00
−0.00 +   +0.00   =  +0.00  except when round minus infinity, then
                     −0.00.
```


6.6.3 Multiplication

Multiplication of floating-point numbers is accomplished by the following steps:

1. Access the operands. If either is a NaN or an infinity, branch to the appropriate special case. If either is zero, then the result is also zero and steps 3, 4, 5 may be omitted. If either is denormal, then normalize it.

2. If both operands' signs are positive or both signs are negative, then the result sign is positive; otherwise it is negative.

3. Add the exponents and subtract the exponent bias to give the preliminary result exponent.

4. Multiply the fractions. If the fraction product is less than 2, then continue; otherwise shift right one place and increment the preliminary exponent by one.

5. If the fraction is inexact, then round.

6. Store the result.

Multiplication of floating-point numbers is subject to the invalid, inexact, overflow and underflow exceptions. Overflow occurs if, after normalization and rounding, the product of two infinite non-zero numbers has an exponent which is greater than +127 in single or +1023 in double. Overflow and underflow may be exact or inexact. Refer to Section 6.9 for details.

Here is an example of a single-precision multiplication:

```
loadi     $0xcf8ccccd, r0
movws     r0, f0
loadi     $0xc27ffee1, r0
movws     r0, f1
muls      f0, f1
stors     f1, place3
place3:   .single
```

After the first **loadi** and **movws**, the contents of f0 is

```
sign      :=    1
exponent  :=    00010011111
fraction  :=    1.00011001100110011001101000000000000000000000000000000
```

After the second **loadi** and **movws**, the contents of f1 is

```
sign      :=    1
exponent  :=    00010000100
fraction  :=    1.11111111111111011100001000000000000000000000000000000
```

Neither operand is zero, denormal, infinite nor NaN. The preliminary exponent is 00010011111 + 00010000100 − single_bias = 00010100100. The sign is 0. The product fraction is

10.00110011001100010011111001001100100001001001101

which is normalized to

1.00011001100110001001111001001100100001001001101

and the exponent is incremented to 00010100101. The fraction is not exact, so rounding is required and the inexact exception is signalled. Suppose the rounding mode is the directed rounding mode, round to plus infinity. Since the sign of the result is positive, the rounded fraction result is

1.00011001100110001010

The result may now be assembled:

```
sign      :=    0
exponent  :=    00010000101
fraction  :=    1.00011001100110001010000000000000000000000000000000000
```

and this is stored in location "place3" as

01000010100011001100110001010000.

The next example shows the result of an invalid multiplication, zero times infinity, illustrating the default NaN result and the signalling of the floating invalid operation exception.

```
loadi     $0x80000000, r0
movws     r0, f0
loadi     $0x7f800000, r0
movws     r0, f1
muls      f0, f1
stors     f1, place3
place3:   .single
```

After the first **loadi** and **movws**, the contents of f0 is

```
sign      :=    1
exponent  :=    00000000000
fraction  :=    0.00000000000000000000000000000000000000000000000000000
```

After the second **loadi** and **movws**, the contents of f1 is

```
sign      :=    0
exponent  :=    00011111111
fraction  :=    1.00000000000000000000000000000000000000000000000000000
```

The operands are identified as being zero and infinite, so the special case is invoked. The product is defined to be the default NaN, which in single-precision is

```
sign      :=    1
exponent  :=    00011111111
fraction  :=    1.11111111111111111111111000000000000000000000000000000
```

Note that the sign is negative because the operands' signs differed. The invalid operation exception is signalled, and if the trap enable bit and the enable floating trap group bits in the PSW are both set, then the invalid operation trap is entered.

6.6.4 Division

Division of floating-point numbers is accomplished by the following steps:

1. Access the operands. If either is denormal, then normalize it. If either operand is zero, infinite or a NaN, then branch to the appropriate special case.

2. If both operands' signs are positive or both signs are negative, then the result sign is positive; otherwise it is negative.

3. Subtract the exponent of A from the exponent of B and add the exponent bias to give the preliminary result exponent.

4. Divide the A fraction into the B fraction. If the fraction quotient is less than 2 and greater than or equal to 1, then continue; otherwise shift left one place and decrement the preliminary exponent by one.

5. If the fraction is inexact, then round.

6. Store the result.

Division of floating-point numbers is subject to the invalid, inexact, divide by zero, overflow and underflow exceptions.

Here is an example of a single-precision division:

```
loadi       $0xc27ffee1, r0
movws       r0, f0
loadi       $0xcf8ccccd, r0
movws       r0, f1
divs        f0, f1
stors       f1, place4
place4:     .single
```

After the first **loadi** and **movws**, the contents of f0 is

```
sign        :=      1
exponent    :=      00010000100
fraction    :=      1.11111111111111011100001000000000000000000000000000
```

After the second **loadi** and **movws**, the contents of f1 is

```
sign        :=      1
exponent    :=      00010011111
fraction    :=      1.00011001100110011001101000000000000000000000000000
```

Neither operand is zero, denormal, infinite nor NaN. The sign of the quotient is 0, since the operands' signs are the same. The preliminary exponent is 00010011111 − 00010000100 + exponent_bias = 00010011010. The quotient is approximately

0.10001100110011010110101011011010010010110010101011

which is normalized to

1.00011001100110101101010110110100100101011001010110

The exponent becomes

00010011001

Suppose the rounding mode is round toward zero. Then

1.00011001100110101101010

So the complete result is

sign	:=	0
exponent	:=	00010011001
fraction	:=	1.00011001100110101101010000000000000000000000000000000

which, when written to "place4", is

01001100100011001100110101101010

Division of a finite non-zero number by zero signals a divide by zero exception. The default result is a properly signed infinity. Division of an inifinite number by zero also yields a signed infinity, but since this is not creating an infinite result out of finite operands, the Standard requires that the divide by zero exception NOT be signalled. Zero divided by zero is undefined, so the result is the default quiet NaN, and the floating invalid operation exception is signalled. A NaN divided by zero yields that same NaN (possibly made quiet), and if the NaN was signalling, the invalid operation exception is also signalled. (This shows that not all divisions with a zero divisor necessarily signal the divide by zero exception.)

Division by infinity is very much like multiplying by zero. In most cases, the result is a properly signed zero. However, infinity divided by infinity, like infinity times zero, is undefined. The default result is a quiet NaN, and the invalid operation exception is signalled.

6.6.5 Comparison

The floating-point comparison operations use two floating-point arguments from the floating-point register file and return two bits to the N and Z condition codes in the PSW. If the b operand equals the a operand, then Z := 1 and N := 0. If b is greater than a, then Z:= 0 and N := 0. If b is less than a, then Z := 0 and N := 1. If either or both of the operands is a NaN, then the unordered condition exists, and Z := 1 and N := 1. Comparison operations never signal any exception, and no result is written into the register file.

6.6.6 Floating-Point Remainder, SquareRoot and Round to Integer

These floating-point operations are mandated in the Standard and must be available to the user. However, their frequency of use does not warrant their implementation directly in the hardware. Instead they are provided by a math library, which also handles trigonometric functions, logarithms and exponentiation.

6.6.6.1 Floating-Point Round to Integer in Floating-Point Format

The purpose of this operation is to round a floating-point number to either of the two nearest integer values, depending on the selected rounding mode.

For any real number, there exists both an integer which is the smallest integer greater than or equal to the given real number, as well as an integer which is the largest integer less than or equal to the given real number. If the real number is already an integer, then both adjacent integers have the same value as the operand. If the two adjacent integers have different values, then the inexact exception is signalled. Depending on the rounding mode and the fractional part of the real number, one of these two adjacent integers is delivered as the result of round to integer.

If the rounding mode is round toward zero, then the smaller in magnitude of the two adjacent integers is delivered to the destination. If the rounding mode is round toward positive infinity, then the more positive value of the adjacent integers is delivered. For round toward minus infinity, the more negative of the adjacent integers is delivered. For round nearest, the distances between the operand and each of the adjacent integers are compared. The smaller distance of the two selects the result. If the real number lies exactly halfway between the adjacent integers, then whichever of the adjacent integers is even (lsb = 0) is selected as the result.

It is not possible for this operation to overflow or underflow, nor to signal the divide by zero exception. The inexact exception is signalled when the operand is not already an integer in floating-point format. If the operand happens to be a signalling NaN, then the floating invalid operation exception will be signalled.

6.6.6.2 Floating-Point Square Root

The square root of +0 is +0. The square root of −0 is −0. The square root of any negative number beside −0 is a NaN with an invalid operation exception. The square root of +infinity is +infinity. The square root of any positive finite non-zero real number is another positive finite non-zero real number.

Square root cannot overflow nor underflow nor divide by zero. (For single, the interesting operands are between $+2^{-149}$ and about $+2^{+128}$. The square roots lie between $+2^{-75}$ and $+2^{+64}$, so no overflow nor underflow may occur. Double is similar.) The result may only be inexact, invalid or O.K.

6.6.6.3 Floating-Point Remainder

The floating-point remainder is defined r := b REM a by the statements

n, the integral quotient, is the nearest integer to b/a
r = b − a x n
| n − b/a | ≤ 1/2
when | n − b/a | = 1/2, then n is even

The integral quotient is discarded, and only r is returned. It is possible for this integral quotient to exceed 2^8 bits long for single-precision and 2^{11} bits long for double-precision. The extraordinary number of quotient bits required means that this operation may take a very long time. Notice that r may be positive or negative or zero, so long as its magnitude is no larger than one half the magnitude of b. For example,

17 REM 3 is −1 (not the −2 which results from an integer mod function)
−17 REM 3 is 1
−17 REM −3 is 1
17 REM −3 is −1

6.7 FLOATING-POINT FORMAT CONVERSION INSTRUCTIONS

It must be possible to convert between single- and double-precision floating-point numbers. It is also convenient to be able to convert between integers and floating-point numbers. Finally, it must be possible to convert ASCII strings of a given form into floating-point values, and convert floating-point numbers to ASCII strings.

All of the following conversions are implemented by CLIPPER instructions. The functional behavior of the instructions are described with examples.

6.7.1 Conversions Between Floating-Point Formats

The CLIPPER provides two instructions for conversions between floating-point formats. The first is **cnvds**, which converts a double-precision operand to a single-precision result. The second is **cnvsd**, which converts a single-precision floating-point number to double-precision.

The Standard is brief in this matter:

> "5.4 Conversion between floating-point and integer formats. It shall be possible to convert floating-point numbers between all supported formats. If the conversion is to a narrower precision, the result shall be rounded as specified in [paragraph] 4. Conversion to a wider precision is exact."

6.7.1.1 Convert Double to Single

The exponent range of double-precision is very much larger than that of single-precision. It is possible for a double-precision value to fall into one of four categories. The first is the values with biased exponent of 2047. The second category includes those values too large to fit into the single-precision exponent range. Values in the range where they can be approximately represented in normalized single-precision format form the third category, and in the last category are values too small to be represented in normalized single-precision.

If the biased exponent of the operand is 2047, then the value is either infinite or a NaN. In either case, the significand of the single result is the significand of the operand truncated to 23 places to the right of the binary point. If the operand is a signalling NaN, then the invalid operation exception is raised. If the biased exponent is between 2046 and (1023 + 127), then the result is an overflow. If the overflow trap is enabled, then nothing further is done to the result, and the trap handler is invoked. Otherwise, depending on the rounding mode, the properly signed max_real or infinity is delivered and the overflow exception is signalled. If the biased exponent is between (1023 − 150) and 0 then the result will underflow. If the underflow trap is enabled, then nothing further is done to the result, and the trap handler is invoked. Otherwise, depending on the rounding mode, the properly signed zero or min_real is delivered. Finally, for an exponent between (1023 + 126) and (1023 − 150) the new biased exponent is the old biased exponent minus 896 (which is 1023 − 127) and the fraction is truncated to 23 bits. If the biased exponent as derived is negative or zero, then the biased exponent is incremented while the significand is shifted right until the biased exponent is +1. Rounding is performed if necessary.

For example,

```
loadi   $0x33333333, r0
loadi   $0x46e33333, r1
movwd   r0, f0
cnvds   f0, f1
stors   f1
```

This code fragment will store -1.2×2^{111} into register f0 as a double-precision value. This is subsequently converted to a single-precision value with rounding required, so the inexact exception is signalled. Assuming round nearest mode, the result is stored to memory as the word 7719999a.

6.7.1.2 Convert Single to Double

Conversion from single- to double-precision always fits, so overflow, underflow and inexact exceptions cannot arise. In any case, the significand of the destination is just the significand of the single-precision source with zeroes appended to the least significant end of the significand. If the single-precision source is a NaN, then the double-precision destination is also a NaN with the significand as described before. If the source is infinite, then the destination is also infinite. If the source is zero, then the destination is double-precision zero with the same sign. All other numbers in single-precision (including denormal numbers) are exactly represented in the double format.

Example:

```
loadi     $0x7fc00000 , r0
movws     r0 , f0
convsd    f0 , f1
stord     f1
```

The code fragment above stores a quiet NaN into f0, then converts it to double-precision. The result stored in memory is

```
low word:    00000000
high word:   7ff80000
```

which is also a quiet NaN.

6.7.2 Conversion Between Floating-Point and Integer Formats

The CLIPPER architecture provides instructions for converting a 32-bit integer (a word in a general register) to either a single- or double-precision floating-point value in a floating-point register.

Conversions from floating-point single- or double-precision to integers are also supported as required by the Standard and the specifications of important high level languages. In particular, in the conversion in ISO PASCAL is performed by the "round" function, which is incompatible with the Standard's rounding definitions.

6.7.2.1 Convert Double Floating to Word: IEEE Standard Version

The **cnvdw** instruction converts from double-precision to 32-bit signed integer format, with rounding according to the rounding mode flags in the PSW.

If the value in the double-precision source is a NaN, an infinity, or otherwise outside the representable range of $+2^{31}-1$ down to -2^{31}, then the invalid operation exception is raised and the result is undefined.

If the value lies within the representable range of the signed integer type but has a fractional part which is not exactly zero, then the inexact exception is raised. Since the destination is a general register and not a floating-point register, the previous contents of the destination register are not saved in the hidden "last overwritten floating destination" register. For example,

```
loadi     $0x20000000, r0
loadi     $0xc1535700, r1
movwd     r0, f0
cnvdw     f0, r2
```

This code fragment creates and loads the real number −5069824.50 into a floating-point register. That value is then converted to an integer. If the rounding mode is round nearest, the result delivered to general register r2 is −5069824 (hex ffb2a400). If the rounding mode had been round toward zero or toward plus infinity, the result would have been the same. For round toward minus infinity, however, the result would have been −5069825 (hex ffb2a3ff).

6.7.2.2 Convert Truncating Double Floating to Word

Independently of the rounding mode bits, the **cnvtdw** instruction converts the floating-point source to a signed integer having the largest magnitude not greater than the magnitude of the source. This may be viewed as expressing the source as a fixed point number and then truncating (setting to zero) all bits to the right of the radix point. This value is stored into the destination general register. Repeating the previous example for this conversion:

```
loadi   $0x20000000, r0
loadi   $0xc1535700, r1
movwd   r0, f0
cnvtdw  f0, r2
```

This code fragment creates and loads the real number −5069824.50 into a floating-point register. That value is then converted to an integer by truncation. The result delivered to general register r2 is −5069824 (hex ffb2a400).

The **cnvtdw** instruction supports the INT function in ANSI FORTRAN 77 and the TRUNC operation in ISO PASCAL. This operation behaves exactly like the IEEE Standard's convert instruction, **cnvdw**, when the rounding mode is round toward zero.

If the value of the result differs from the value of the source, the inexact exception is signalled. If the source is a NaN, infinite, or outside of the range of signed integer type, then the invalid operation exception is signalled.

6.7.2.3 Convert Rounding Double Floating to Word

Although not described in the Standard, the **cnvrdw** instruction is required to support certain high level languages, for example, ISO PASCAL, which round a real number to integer using magnitude biased rounding. The Standard makes no provision for biased rounding, choosing unbiased rounding instead. Biased magnitude rounding is defined in the ISO PASCAL Standard as follows:

> "round(x) From the expression x that shall be of real-type, this function shall return a result of integer-type. If x is positive or zero, round(x) shall be equivalent to trunc(x+0.5), otherwise round(x) shall be equivalent to trunc(x−0.5). It shall be an error if such a value does not exist."

If the value of the result differs from the value of the source, then the inexact exception is signalled. If the source is a NaN, infinite, or outside of the range of signed integer type, then the invalid operation exception is signalled.

Repeating the previous example:

```
loadi   $0x20000000, r0
loadi   $0xc1535700, r1
movwd   r0, f0
cnvrdw  f0, r2
```

This code fragment creates and loads the real number −5069824.50 into a floating-point register. That value is then converted to an integer by biased magnitude rounding. The result delivered to general register r2 is −5069825 (hex ffb2a3ff).

6.7.2.4 Convert Single Floating to Word: IEEE Standard Version

The **cnvsw** instruction converts from single-precision to 32-bit signed integer format, with rounding according to the rounding mode bits in the PSW.

If the value in the single-precision source is a NaN, an infinity, or otherwise outside the representable range of $+2^{31}-1$ down to -2^{31}, then the invalid operation exception is raised and the result is undefined. If the value lies within the representable range of the signed integer type but has a fractional part (when considered as a fixed point real number) which is not exactly zero, then the inexact exception is raised. For example,

```
loadi   $0xc2142000, r1
movws   r1, f0
cnvsw   f0, r2
```

This code fragment creates and loads the real number −37.03125 into a floating-point register and then converts that value to an integer. If the rounding mode is round nearest, then the result delivered to general register r2 is −37 (hex fffffffdb). Had the rounding mode been round toward zero or toward plus infinity, the result would have been the same. For round toward minus infinity, however, the result would have been −38 (hex fffffffda).

Since the destination is a general register and not a floating-point register, the previous contents of the destination register are not saved in the "last overwritten floating destination" register.

6.7.2.5 Convert Truncating Single Floating to Word

Independently of the rounding mode bits, the **cnvtsw** instruction converts the floating-point source to a signed integer having the largest magnitude not greater than the magnitude of the source. This may be viewed as expressing the source as a fixed point number and then truncating (setting to zero) all bits to the right of the radix point. The result is stored into the destination general register.

Repeating the previous example for this conversion:

```
loadi    $0xc2142000, r1
movws    r1, f0
cnvsw    f0, r2
```

This code fragment creates and loads the real number −37.03125 into a floating-point register. That value is then converted to an integer by truncation. The result delivered to general register r2 is −37 (hex fffffffdb).

The **cnvtsw** instruction supports the INT function in ANSI FORTRAN 77 and the TRUNC operation in ISO PASCAL. This operation behaves exactly like the Standard's convert instruction, **cnvdw**, when the rounding mode is round toward zero.

If the value of the result differs from the value of the source then the inexact exception is signalled. If the source is a NaN, infinite, or outside of the range of signed integer type, then the invalid operation exception is signalled.

6.7.2.6 Convert Rounding Single Floating to Word

Although not described in the Standard, the **cnvrsw** instruction is required to support certain high level languages (for example, ISO PASCAL) which round a real number to an integer using magnitude biased rounding. The Standard makes no provision for biased rounding, choosing unbiased rounding instead. Biased magnitude rounding is defined in the ISO PASCAL Standard as follows:

> "round(x) From the expression x that shall be of real-type, this function shall return a result of integer-type. If x is positive or zero, round(x) shall be equivalent to trunc(x+0.5), otherwise round(x) shall be equivalent to trunc(x−0.5). It shall be an error if such a value does not exist."

If the value of the result differs from the value of the source, the inexact exception is signalled. If the source is a NaN, infinite, or outside of the range of signed integer type, then the invalid operation exception is signalled.

Repeating once again the previous example:

```
loadi    $0xc2142000, r1
movws    r1, f0
cnvsw    f0, r2
```

This code fragment creates and loads the real number −37.03125 into a floating-point register. That value is then converted to an integer by biased magnitude rounding. The result delivered to general register r2 is −37 (hex fffffffdb).

6.8 IEEE 754 APPENDIX OPERATIONS

There are a number of functions described in the Appendix to the IEEE Standard. Although these functions are not required (because they are not a part of the Standard itself), many users will want to have access to these functions from time to time. The descriptions which follow make clearer which of these functions are directly available in the form of instructions. The other functions are described, but the user must provide the code to implement them.

6.8.1 Floating-Point Scale By an Integer Power of Two

The scalb(y,N) function of the Appendix is provided by the **scalbs** and **scalbd** instructions.

scalbs takes as arguments an integer and a single-precision floating-point value. The floating-point argument is effectively multiplied by 2 to the power i, where i is the value found in the integer argument. If the floating-point argument is a NaN, zero, or infinite, then the result is just this argument. If the single argument is normalized, then i is added to the exponent. If the exponent is now too large, overflow occurs. If the exponent is now too small, underflow occurs. Otherwise, the result is returned without having to do a multiply. If the single operand is denormal, then the significand field is normalized, the exponent is adjusted, and then i is added to the exponent. Again, if this is too big or too small, overflow or underflow occurs.

scalbd behaves analogously to **scalbs**, but uses a double-precision floating-point argument.

6.8.2 Negate Floating Register

Two instructions, **negs** and **negd**, are provided. Each of these complements the sign of the floating-point argument. **negs**(x) differs from 0 − x when x is positive zero and the rounding mode is not toward minus infinity. **negs** yields −0, while 0 − x yields +0 for round toward zero, toward plus infinity, or toward nearest.

6.8.3 Copy Sign

The function copysign(x,y) returns x with the sign of y. This is a trivial bit manipulation on the CLIPPER. The sign bit of x is masked off, and all bits but the sign bit of b are masked off. The two results are ORed together and this is returned. Because of the CLIPPER floating-point architecture, a double-precision version of this function will work properly on single-precision operands as well.

6.8.4 Get Binary Exponent

For a single- or double-precision floating-point number x, logb(x) returns the unbiased exponent of x as a signed integer in the same format as x for reasonable values of x. The special cases are NaNs, which are returned unchanged (or made quiet with a invalid operation exception if signalling), plus or minus infinity which return +infinity, and plus or minus zero which returns −infinity and signals the divide by zero exception. Two versions of this function are needed, one for single and another for double.

6.8.5 Successor Function

For a pair of floating-point numbers x and y, the function nextafter(x,y) returns the next representable neighbor of x in the direction toward y. There are two special cases:

1. x = y.	In this case, x is returned and no exception is signalled.
2. x or y is a NaN.	If one of them is signalling, then the floating invalid operation exception is signalled. In any case the result is one of the NaNs.

If x is ±max_real and nextafter is infinite, then the overflow and inexact exceptions are signalled. If the result of nextafter is a denormal number or zero, then underflow and inexact exceptions are signalled.

6.8.6 Classification Functions

There are several functions listed in the Appendix which allow one to determine whether a given floating-point number is finite, NaN, etc, or provide additional comparison predicates. All of these may be implemented in the obvious ways, using the comparison instructions and a few well chosen constants.

6.9 FPU EXCEPTIONS

Exceptional operands and operation results are handled in conformity with the requirements of the IEEE standard. The exceptions which can occur in the FPU are integer overflow, integer divide by zero, floating inexact result, floating invalid operation, floating overflow, floating underflow and floating divide by zero.

6.9.1 Integer Exceptions

The integer exceptions are signalled by the FPU when an integer operation is performed and the result is not accurately representable as a single word (32-bit) integer.

6.9.1.1 Integer Divide by Zero

The integer divide by zero exception is signalled when an integer divide or mod instruction is executed with zero divisor ((w1) = 0.) Integer divide by zero cannot give any acceptable integer as its result, since the only remotely satisfying answer would be an infinite value, and there exists no such value among the 32-bit integers.

Because there is no associated trap enable bit nor associated exception flag for this exception, it always causes a trap.

6.9.1.2 Integer Overflow

Integer overflow signifies that some bits with higher significance than the msb of the destination format (signed or unsigned) have been lost. The loss of left hand significant bits is a serious matter under most circumstances, but if the result is useful mod some power of 2 (especially if it is useful mod 2^{32}), then the lost left hand bits are safely discarded. Specifically, integer overflow is signalled for the following conditions:

1. Unsigned integer multiply, when the product is greater than $2^{31} - 1$.
2. Signed integer multiply, when the product is less than -2^{30} or greater than $+2^{30} - 1$.
3. Signed integer divide or mod, when the dividend is -2^{30} and the divisor is -1.

This exception sets the V condition code in the PSW. There is no integer overflow trap, and thus no need for an integer overflow trap enable flag.

6.9.2 Floating-Point Exceptions

The FPU supports all floating-point exceptions in the manner called for by the IEEE Standard 754. The floating-point exceptions are invalid operation, inexact result, overflow result, underflow result, and divide by zero.

The five floating-point exceptions are signalled in cases where the result of a computation is a value which cannot be represented accurately in the destination floating-point format.

For each exception there corresponds a set of two important bits in the PSW. The first of these is a trap enable bit, and the second is a "sticky" exception flag. Each exception flag is set any time the corresponding exception occurs in a floating-point operation. Since these flags are in the PSW, it is straightforward for the user to save and restore them, test them individually or in groups, and set or clear them individually or in groups.

The five floating-point exceptions each can be enabled or disabled by setting or clearing their individual trap enable bits. In addition, the five floating-point traps have a group trap enable bit, EFT. If the corresponding trap enable bit is true and the group trap enable bit is also true when a floating-point exception is signalled, then a trap occurs. If the trap enable bit is false or the group trap enable is false, then the trap is not taken and execution proceeds as if the exception had not happened, but with a possibly different answer. When an exception is signalled and the corresponding enable bit is false, no trap is taken

and the value returned to the destination is only a "best representable guess" at the true result. For example, on overflow or underflow, the result will be "wrapped" if the trap enable bit is true; but if the trap enable is false, then the result is a signed infinity or max_real in the case of overflow, and denormal or zero in the case of underflow. The EFT bit is very useful for carefully constructed numeric codes which are capable of resolving exceptions internally and do not want the overhead of system calls for any exceptions which may occur, but which nevertheless need the value which would be given to the trap handler in such a case.

Regardless of whether or not the trap is taken, the "sticky flag" corresponding to that specific exception is set to a one. In this way it is possible for a user to determine whether, for example, any overflow or underflow has occurred in a given stretch of instructions. He would do this by clearing the sticky flags for underflow and overflow in the PSW, executing the code, then observing these same bits in the PSW. This makes a difference for some types of programs.

The IEEE 754 standard for binary floating-point arithmetic requires that the floating-point unit (from the perspective of the software user) provide different results in some cases depending on whether or not the particular exception has its trap enabled. In particular, on underflow without a trap, the value is made into a denormal number or zero if it is very tiny, while with the trap enabled, the delivered result is normalized with the exponent wrapped. For overflow the situation is similar, with an exponent wrapped result if the trap is enabled, and a signed infinity or max_real if the trap is disabled. Invalid, Inexact and Zero Divide behave independently of their respective trap enables.

6.9.2.1 Floating Overflow

The floating overflow exception is signalled when the biased exponent of the result (after rounding) is greater than the largest finite representable exponent. With addition and subtraction, overflow occurs when two large numbers are added (at least one of them must have a biased exponent of +126 (single-precision) or +1022 (double-precision) and the fraction addition (or the subsequent rounding) has a carry out of the msb position. The overflow may coincide with the fraction sum being inexact, though this is not necessarily the case. With multiplication, overflow occurs if, after normalization and rounding, the product of two finite non-zero numbers has an exponent greater than +127 (single-precision) or +1023 (double-precision). Overflow for multiplication may be exact or inexact.

If the EFV flag is set, the computed result is delivered to the destination with the normalized rounded fraction of the true result (though the delivered exponent is usually wrong because of missing additional leading bits in the exponent field). For single precision overflows, if the biased exponent of the true result is 255, then biased exponent 255 is delivered to the destination. If the true biased exponent of the result is 256..408, then the true biased exponent minus 256 is delivered to the destination. (Note that this is not the exponent wrapped result called for by the IEEE 754 specification; the wrap must be adjusted by system software before delivery to a user's trap handler.) For double-precision, the overflow exponents (biased) lie in the range 2047..3120. These are mapped to 2047 and 0..1072 respectively. These must be adjusted by $(3/4) \times 2^{11}$ (1536) to obtain the IEEE Standard wrapped exponent.

If the EFV flag is clear, then the computed result is discarded, and the properly signed default value (infinity or max_real, depending on rounding mode) is delivered to the destination. (max_real is the maximum representable value in the given floating-point format: single_max_real = $2^{128} - 2^{104}$; double_max_real = $2^{1024} - 2^{971}$.) The floating inexact exception is also signalled. If the rounding mode is round toward zero, the value delivered to the destination is the maximum finite representable number (max_real) with the appropriate sign. If the rounding mode is round toward +inf, then a positive signed overflow is replaced with +inf, while a negative signed overflow is replaced by minus max_real. For round toward −inf, a positive overflow goes to plus max_real, while a negative overflow goes to −inf.

6.9.2.2 Floating Underflow

The conditions under which the floating underflow exception condition is signalled differ according to the setting of the EFU flag. If EFU is set, the floating underflow exception is signalled when the result of an operation (before rounding) has a biased exponent less than the minimum representable biased exponent for a normalized number. If the true biased exponent of the result is 0, then biased exponent zero is delivered to the destination. If the true biased exponent is less than zero, then the exponent delivered to the destination is true biased exponent plus 256 (2048 for double.) To conform to the Specification, the exponent must be adjusted by system software before delivery to the programs's trap handler. The range of underflowed biased exponents for single-precision is 0..−275; for double-precision the range is 0..−1125.

If the EFU flag is clear, then the underflowed fraction is right shifted as the exponent is incremented until the biased exponent equals one. At this point, the result has been restated as a denormal number. If this representation is exact, then no under-flow exception is signalled. If the representation is inexact, then the result is rounded and delivered to the destination, and both the inexact and underflow exceptions are signalled.

6.9.2.3 Floating Divide By Zero

The floating divide by zero exception is signalled when the divisor is zero and the dividend is non-zero and finite. (If the dividend is zero, the result is the default quiet NaN (all ones in the fraction and exponent fields) and the FI flag is set. If the dividend is infinite, the result is infinite and no exception flags are set.) The result delivered to the destination is a correctly signed infinity.

6.9.2.4 Floating Invalid Operation

The floating invalid operation exception is signalled in the following cases:

1. One of the operands is a signalling NaN

2. Add/Subtract, magnitude subtraction of infinities:

	(+inf)	−	(+inf)
or	(+inf)	+	(−inf)
or	(−inf)	−	(−inf)
or	(−inf)	+	(+inf)

3. Multiplication

	0	×	inf
or	inf	×	0

4. Division

	0	/	0
or	inf	/	inf

The value written to the destination is always a NaN. The NaN is either the NaN operand (the second operand if both are NaNs) made quiet if it was signalling (by setting the msb of the explicit fraction field), or the default NaN created by the hardware. The default NaN is quiet, and its fraction field is all ones.

6.9.2.5 Floating Inexact

The inexact exception is signalled in the following cases:

1. The result of an operation overflows and the overflow trap is disabled.
2. The result of an operation cannot be exactly represented in the precision of the destination. In this case, the result is rounded according to the rounding mode specified in the PSW so that it has the precision of the destination, and the rounded result is then delivered to the destination.

When the inexact exception is signalled, the result is always rounded regardless of whether or not the trap is taken.

Chapter 7
CLIPPER Bus

7.1 INTRODUCTION

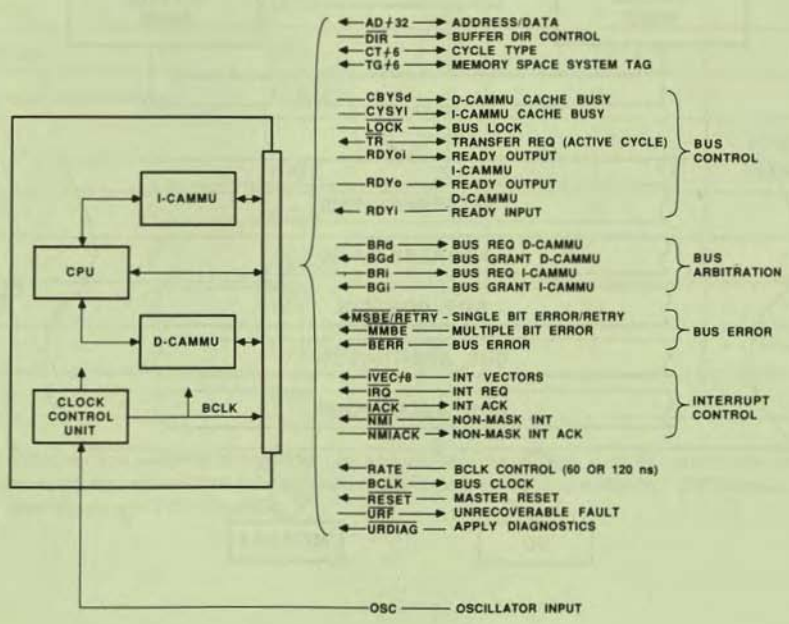
A traditional limitation in computer system performance has been the system bus. This is because all information exchanged between the CPU, memory, and I/O must be transferred on the bus according to the specific bus protocol, which can add significant overhead to each bus operation. Consequently the CPU, which may execute most instructions in one or two clock cycles, often has to wait for completion of bus operations which require several more cycles to complete.

The CLIPPER Module addresses the bus limitation problem in two ways. First, the module incorporates many architectural features, such as caching, pipelining, and instruction prefetch, which minimize system bus bottleneck and increase the effective bandwidth of the bus. These features have already been discussed in previous chapters. Second, the module utilizes a high-speed bus which is designed to optimize the performance of CLIPPER-based systems. This is the CLIPPER Bus, and is the topic of this chapter.

7.2 Signal Descriptions

The CLIPPER Bus is a high-speed, synchronous bus defined by the seventy-three (73) signal lines extending from the CLIPPER Module via its 96-pin connector (see Figures 7.1 and 7.2). These signals are buffered, if required, to form the

Figure 7.1 CLIPPER Bus Signals



CLIPPER system bus, referred to as the CLIPPER Bus. All module outputs except BCLK are open drain and are tied to pullup resistors located within the module. BCLK is CMOS compatible; all other signals are TTL compatible.

Table 7.1 contains a list of the CLIPPER Bus signals, and descriptions of their drive characteristics. Appendix A contains brief descriptions of the signals, which can be grouped as follows:

- 32-bit Address/Data multiplexed lines used for address and data transfer.
- Clocks.
- Cycle Type signals used to identify the number of bytes or words transferred during a bus operation, to identify the operation as a read or a write, and to identify the bus master executing the operation as a CPU or I/O device.
- Memory Space System Tags used to identify the address space being accessed, and the caching policy which applies to the accessed data.
- Bus arbitration signals.
- Bus control signals.
- Interrupt bus and control signals.
- Error signals.
- Reset and diagnostics control.

These signals fulfill the need for system bus efficiency, speed, and flexibility, as explained in the following sections.

Figure 7.2 CLIPPER Bus

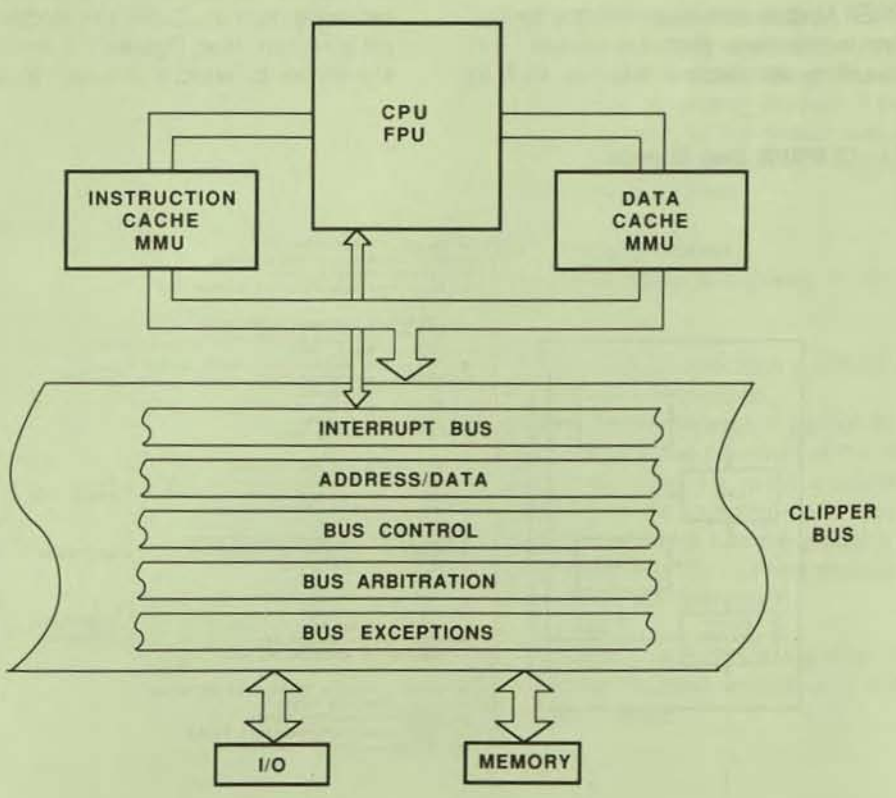


TABLE 7.1 Signal Summary

Signal	Mnemonic	Input/Output	Active State
Address/Data Bus	AD	I/O	HIGH
Buffer Direction Control	$\overline{DIR}$	O	LOW
Memory Space System Tag	TG	I/O	HIGH
Cycle Type	CT	I/O	HIGH
Cache Busy	CBSYi, CBSYd	O	HIGH
Bus Lock	$\overline{LOCK}$	O	LOW
Transfer Request	$\overline{TR}$	I/O	LOW
Ready	RDYi	I	HIGH
	RDYo, RDYoi	O	HIGH
Busy Request	BRi, BRd	O	HIGH
Bus Grant	BGi, BGd	I	HIGH
Memory Single Bit Error/Retry	$\overline{MSBE/RETRY}$	I	LOW
Memory Multiple Bit Error	$\overline{MMBE}$	I	LOW
Bus Error	$\overline{BERR}$	I	LOW
Interrupt Vector Bus	$\overline{IVEC}$	I	LOW
Interrupt Request	$\overline{IRQ}$	I	LOW
Interrupt Acknowledge	$\overline{IACK}$	O	LOW
Non-Maskable Interrupt	$\overline{NMI}$	I	LOW
Non-Maskable Interrupt Acknowledge	$\overline{NMIACK}$	O	LOW
BCLK Rate Select	RATE	I	HIGH=120 ns LOW=60 ns
Bus Clock	BCLK	O	—
Master Reset	$\overline{RESET}$	I	LOW
Unrecoverable Fault	$\overline{URF}$	O	LOW
Apply Diagnostics	$\overline{URDIAG}$	I	LOW
Oscillator Input	OSC	I	–

1. Inputs are designed with a nominal switching threshold of 1.3v and are therefore referred to as TTL compatible. All outputs (excluding BCLK) are open drain structures with external pull-up resistors (220Ω) to V_{CC}. BCLK is a standard CMOS output structure. BCLK is CMOS compatible. All other signals are TTL compatible.

7.2.1 Address/Data Bus

An address/data bus within the CLIPPER Bus, called the AD Bus, is used for addressing and data transfer between the CLIPPER Module, memory, and I/O. The AD Bus consists of 32 open drain, multiplexed lines which transfer the address during the first two cycles of each bus operation, then data. The bus is active high, positive logic. The AD lines tie directly to each CAMMU.

7.2.2 Clocks

The CLIPPER Module is clocked by an external oscillator signal applied at the module OSC signal connector located within the module. Attached to the connector is a small-diameter coaxial cable for signal insertion. The OSC signal must be TTL compatible, with a duty cycle within 60/40 to 40/60.

A Clock Control Unit within the CLIPPER Module uses OSC to generate MCLK and BCLK. MCLK clocks the CPU and CAMMUs, and is 1/2 the frequency of OSC. BCLK is the CLIPPER Bus master clock, and is either 1/2 or 1/4 the clock frequency of MCLK depending on the user-selectable state of the CLIPPER Bus RATE signal. For an OSC frequency of 66.67 MHz, therefore, MCLK frequency is 33.33 MHz, and BCLK frequency is either 16.67 MHz (60 ns) or 8.33 MHz (120 ns).

All signals on the CLIPPER Bus are sampled on the rising edge of BCLK, resulting in a synchronous bus.

7.2.3 Cycle Type

The Cycle Type signals, CT<5:0>, are asserted by CLIPPER Bus masters to identify the width of data being transferred, to identify bus operations as either reads, writes, or global writes, and to identify the bus masters as either CPU or I/O devices.

Signal CT<5> identifies the bus master as an I/O when HIGH, and as a CPU when LOW. This signal supports Bus Watch operation, which invokes Watch CPU Writes, Watch I/O Writes, and Watch I/O Reads according to the signal state.

CT<4> is the traditional system bus read/write signal, identifying the bus operation as a read or write operation. When CT<4> is HIGH, the bus master is executing a read operation.

CT<3:2> identify whether the operation is a quadword transfer, a word/halfword/byte transfer, or a global word write as follows:

CT<3:2> States	Operation
00	word/halfword/byte transfer
01	quadword transfer
10	reserved
11	global write

Quadword transfers are an efficient way to move data, and are particularly well-suited for data transfers between the four-word CAMMU cache lines and memory. During quadword transfers, four words are transferred to or from sequential memory locations with only one arbitration for the bus, and only one address asserted. AD<3:0> of the address must be 0, pointing to a quadword boundary. All CAMMUs support quadword transfers.

Global writes allow a bus master to simultaneously write to all CAMMUs in a system for fast configuration and control, and is primarily intended for use in systems utilizing more than one CLIPPER. In such systems, a CLIPPER CPU can, in one operation, write to a particular register or execute selective reset in all CAMMUs other than its companion D-CAMMU. Global writes are word transfers.

Word/halfword/byte transfers are further defined by CT<1:0> as follows:

CT<1:0> States	Operation
00	whole word transfer
01	reserved
10	byte transfer
11	halfword transfer

AD<1:0> must point to appropriate data boundaries during each of these transfers and must therefore be 0 during whole word transfers, and 0

or 2 during halfword transfers. During byte transfers, AD<1:0> will be in one of the four possible states to point to a byte.

This wide range of data transfer modes offers the systems designer flexibility in interfacing the CLIPPER Module to memory and I/O.

7.2.4 Memory Space System Tags

Associated with each page in memory is a Memory Space System Tag which identifies the address space being accessed, and determines the caching strategy to be used during access of the page. Each tag can also identify cache purge and slave I/O operation as follows:

TG2	TG1	TG0	Description
0	0	0	Main memory, private cacheable, write-through
0	0	1	Main memory, shared cacheable, write-through
0	1	0	Main memory, private cacheable, copy-back
0	1	1	Main memory, noncacheable
1	0	0	I/O space, noncacheable
1	0	1	Boot space, noncacheable
1	1	0	Cache Purge
1	1	1	Slave I/O mode, main memory, noncacheable

These tags are decoded from the two-bit UST field of the CAMMU Control Register (unmapped mode, with TG2=0), or are selected during address translation from the CAMMU Hardwired TLBs or the page tables located in main memory for assertion on CLIPPER Bus lines TG<2:0>.

Other bus masters must also assert the Memory Space System Tags on the bus during their bus operations.

The Memory Space System Tags determine caching policy by identifying an accessed page as cacheable or non-cacheable, private or shared, write-through or copy-back. This feature allows the operating system the flexibility of assigning caching policy on a per page basis.

The tags also identify the real address space of each accessed page as main memory, Boot, or I/O. This feature supports many existing system architectures and greatly increases the effective addressing capability of the CLIPPER Module by allowing three separate 4G byte address spaces. However, no I/O restrictions are placed on the systems designer: the designer can still place I/O in main memory space.

Cache Purge forces purging of cache lines that are hit during writes. This feature facilitates re-use of pages by allowing purging of data belonging to a replaced page which is left in a cache. In multiple CLIPPER Module applications, for instance, a page might be replaced which may leave unpurged data in a cache of the module not initiating the page replacement. The CAMMU initiating the page replacement can invalidate that cache data by writing to the cached data locations using the Cache Purge tag. Watch CPU Writes must be enabled in the CAMMU whose cache is to be purged (see Chapter 9 for program examples using Cache Purge).

The CLIPPER Module Slave I/O operation in effect allows the module to act as a DMA controller. It supports transfer of data between I/O and main memory in DMA-like fashion, but is not intended to replace DMA controllers.

During Slave I/O operation, the CLIPPER Module accesses an individual word, halfword, or byte from a source (such as main memory) which is simultaneously read by a destination device. Both read and write operations can be tagged Slave I/O mode.

Slave I/O read operations are used to transfer data from main memory to an I/O device. The CLIPPER Module executes a read from memory with TG=7, and the memory responds with the data which is read by the I/O but can be ignored by the CPU. The I/O must recognize TG=7 as Slave I/O mode. Timing for the Slave I/O read operation is the same as for a standard read.

Slave I/O write operations are used to transfer data from I/O to main memory. The CLIPPER

Module executes a write to memory with TG=7 using arbitrary data that will not be asserted on the bus. The I/O must recognize TG=7 as slave I/O mode and assert data on the bus. The main memory asserts RDYi to terminate the operation.

Timing for the slave I/O write operation is the same as for a standard write, with the exception that $\overline{DIR}$ transitions after the address phase of the operation as if for a read. This prevents the bus transceivers from driving the bus (from the CLIPPER module) during the data transfer phase of the operation, allowing the I/O to send data without bus contention.

7.2.5 Bus Arbitration Signals

The CLIPPER Bus is a system bus designed for use by many bus masters. Since only one bus master is allowed control of the bus at any time, bus arbitration signals have been included as part of the bus which are used for orderly assignment of the bus to individual masters. These signals are Bus Request I-CAMMU (BRi), Bus Grant I-CAMMU (BGi), Bus Request D-CAMMU (BRd), and Bus Grant D-CAMMU (BGd).

Each CAMMU acquires the bus by first asserting its respective bus request signal. It must then wait until its respective bus grant signal is asserted by system logic, indicating that no other bus master has control of the bus and that the requesting CAMMU has the bus. The CAMMU holds its Bus Request signal asserted until it has completed its bus operation.

Just as the I- and D-CAMMUs have assigned bus-request/bus-grant signals pairs, each other bus master tied to the CLIPPER Bus must be assigned its respective bus arbitration signal pair. This signal pair can either be inherent in the I/O device, or can be user-designed.

The systems designer has full freedom in the design of the actual bus arbitration logic. The logic must simply look at the bus request signal from each bus master, arbitrate simultaneous requests for the bus, and grant the bus to one of the requesting bus masters when the bus is free by asserting the appropriate bus grant signal according to priority best suited for the system.

7.2.6 Bus Control Signals

The bus control signals indicate CLIPPER Bus operation status, which is used to implement bus protocol and control, support Bus Watch, and give a bus master the means to secure the bus indefinitely in order to complete multiple bus operations.

A Ready Input (RDYi) tied to each CAMMU is used to synchronize data transfers between CLIPPER, I/O, and memory. When a CAMMU reads data, it holds the bus in a read state until the responding device asserts RDYi, indicating that the data to be read is on the bus. When a CAMMU writes data, it provides data on the clock following the address phase of the operation until the device being written to asserts RDYi, indicating that it has latched the data. RDYi is thus used to accommodate various response times of devices on the bus. This eliminates the need to introduce for all data transfers the number of wait states necessary to accommodate the slowest device on the bus.

Ready Out (RDYo) is asserted by the D-CAMMU during Bus Watch operation in response to an I/O read of memory data that is cached. RDYo is active only during this operation. If the memory page being read is tagged as a copy-back page, then changes to the page data in the cache are not copied to the page in main memory until the page is replaced by the operating system. If an I/O device reads data from memory that is cached, and if the cache has updated data that has not been copied to the memory location being read, the D-CAMMU asserts Ready Out while asserting Cache Busy. This aborts assertion of memory data. (The memory interface must be designed to abort the memory operation when both Cache Busy and Ready Out are asserted.) The D-CAMMU instead asserts the updated cache data on the CLIPPER Bus, which is read by the I/O device, and RDYo. In this way, transfer of valid data to I/O devices is assured.

Ready Out I-CAMMU (RDYoi) is asserted by the I-CAMMU to indicate assertion or latching of data in response to access by the D-CAMMU. Since only the D-CAMMU can access the I-CAMMU, this signal is tied only to the D-CAMMU.

Two CACHE Busy signals, one for the I-CAMMU (CBSYi) and one for the D-CAMMU (CBSYd), are used to indicate CAMMU internal operations associated with Bus Watch. When a CAMMU Bus Watch mode is invoked during a memory access, the affected CAMMU asserts Cache Busy to indicate that it is checking whether the accessed data location is cached (see Figures 7.8 and 7.14). If the bus operation is a write, the memory interface must not assert Ready Input until Cache Busy is released by the CAMMU. This ensures that the CAMMU has time to update data in its cache before the bus operation is completed. If the operation is a read, the memory interface must not assert data on the Address/Data bus until Cache Busy is released. This allows the CAMMU to abort assertion of data by the memory interface, and to provide cached data if required (see Ready Out description).

Transfer Request ($\overline{TR}$) is asserted by bus masters to indicate that CLIPPER Bus operations are in progress (acts as an active cycle for the operation).

Lock ($\overline{LOCK}$) is used in dual-bus applications in which the CLIPPER Bus is interfaced to a separate I/O bus through a bus adapter or a dual port memory to provide CLIPPER Bus masters with a means of maintaining control of the I/O bus or dual port memory throughout successive bus operations. Lock becomes active during DTU page table access, cache line replacement, and read-modify-write operations.

Direction Control ($\overline{DIR}$) is used to control drive direction of TTL transceivers buffering the CLIPPER Module Address/Data bus. This signal controls the transceivers with proper timing for all bus operations, eliminating the need for such logic in the system.

7.2.7 Interrupt Bus And Control

Interrupts are requests to the CPU from devices on the system bus, typically I/O, to service particular conditions. A disk controller might assert an interrupt, for example, to request reading of its memory buffer by the CPU.

Two types of interrupts are possible: maskable, and non-maskable.

Maskable interrupts can be ignored by simply clearing the EI flag in the SSW. These interrupts are usually assigned to conditions which can be temporarily deferred by the CPU without catastrophic impact on system performance. This gives the system software the option of delaying servicing of these interrupts during critical operations.

Non-maskable interrupts cannot be ignored by the CPU. The non-maskable interrupt is usually assigned to conditions which require immediate servicing in order to prevent loss of data, for example.

The CLIPPER Bus supports both maskable and non-maskable interrupts. Interrupt Request ($\overline{IRQ}$) is the maskable interrupt request, and Non-Maskable Interrupt ($\overline{NMI}$) is the non-maskable interrupt request. Corresponding to these interrupt requests are Interrupt Acknowledge ($\overline{IACK}$) and Non Maskable Interrupt Acknowledge ($\overline{NMIACK}$). The CLIPPER Module CPU asserts the appropriate acknowledge signal in response to an interrupt request.

A separate 8-bit bus within the CLIPPER Bus, the Interrupt Vector Bus ($\overline{IVEC}$<7:0>), is used to transfer a vector number to the CPU when $\overline{IRQ}$ is asserted. The $\overline{IVEC}$ line values are inverted when they are read by the CPU; thus, an $\overline{IVEC}$ value of 00 would be interpreted as FF by the CPU.

The vector number contains both an interrupt level number ($\overline{IVEC}$<7:4>), and an interrupt number ($\overline{IVEC}$<3:0>). There are 16 interrupt levels, which are prioritized with level 0 having the highest priority. A mask number, called the Interrupt Level, can be assigned in bits <7:4> of the CPU System Status Word, which allows the CPU to ignore all interrrupts with level number higher than the Interrupt Level mask. There are 15 possible interrupt numbers within level 0, and 16 possible interrupt numbers within levels 1:15. However, only interrupt levels are prioritized:

interrupt numbers within each level are of equal priority. Level 0 Number 0 vectors to the $\overline{NMI}$ interrupt handler.

Transferring the vector numbers on the Interrupt Vector bus rather than the Address/Data bus speeds CPU response to interrupts—no bus arbitration is required to transfer the numbers. This feature also increases the effective bandwidth of the Address/Data bus by removing interrupt traffic from the bus.

7.2.8 Error Signals

The CLIPPER Bus offers effective means of error reporting and recovery with the Memory Single Bit Error/Retry ($\overline{MSBE}/\overline{RETRY}$), Memory Multiple Bit Error ($\overline{MMBE}$), Bus Error ($\overline{BERR}$), and Unrecoverable Fault ($\overline{URF}$) signals.

Two types of system errors are most common: memory data errors, and bus errors.

Memory data errors are reported with the Memory Single Bit Error/Retry ($\overline{MSBE}/\overline{RETRY}$) and the Memory Multiple Bit Error ($\overline{MMBE}$) signals. These signals are asserted by error detection and correction logic within the memory interface to indicate that a single bit error has been detected and corrected ($\overline{MSBE}/\overline{RETRY}$), and that an uncorrectable multiple bit error has occured ($\overline{MMBE}$). The signals, tied directly to the CAMMUs for fast response, force traps to error-handling routines. Timing for $\overline{MSBE}/\overline{RETRY}$ and $\overline{MMBE}$ is shown in Figure 7.4. Note that the signals are asserted during the same cycle that RDYi is active.

In cases of memory errors, the CAMMU Fault Register does not capture the addresses causing the errors. It is therefore necessary to design an address snapshot register into the system to capture addresses for use by the trap routines servicing the errors, if the addresses are required.

The $\overline{MSBE}/\overline{RETRY}$ signal is also used to abort and retry CLIPPER Bus operations. If the signal is asserted during access of I/O space (TG=4) while RDYi is inactive, the current bus operation is aborted and retried with no trap assertion (see Figure 7.21). This feature is intended to resolve deadlock in dual bus systems, which is discussed in Chapter 8, Hardware Design Considerations.

The Memory Single Bit Error/Retry signal, therefore, operates as follows:

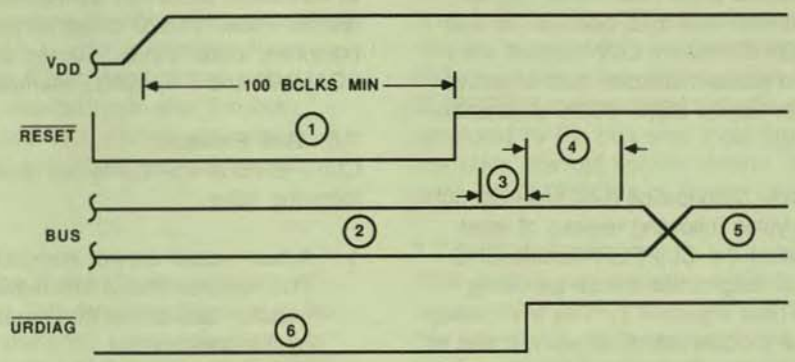
1. If the signal is asserted during any time other than access of I/O space and during the same clock cycle that RDYi is active, the signal reports a corrected memory single bit error. This causes the CAMMU to generate a trap to the CPU.
2. If the signal is asserted during access of I/O space (TG=4) while RDYi is inactive, the current bus operation is aborted and retried by the master CAMMU with no trap assertion to the CPU.

Bus errors are errors associated with bus operations. A common bus error is bus timeout, which results when a bus slave fails to respond to a bus master within a defined time. When such errors occur, the Bus Error ($\overline{BERR}$) signal should be asserted, which forces an abort of the current bus operation causing the error.

Bus Error is tied directly to each CAMMU. If a bus error occurs while a CAMMU is bus master, the CAMMU aborts the bus operation and releases the bus, but does not report the error condition to the CPU. Bus Error should therefore be interfaced to an interrupt controller to force an interrupt to a bus error service routine when bus errors occur.

Some errors allow no clean means of recovery for continuation of program execution. These errors (unrecoverable faults) include the occurrence of a trap during execution of INTRAP or **reti**, and the detection of a fault during self-test (see Section 7.2.9). A trap during execution of INTRAP or **reti** can be avoided by making sure that the Exception Vector Table is set up prior to the occurrence of a trap condition, and that the supervisor stack pointer always points to a valid page. No other conditions generate an unrecoverable fault.

Figure 7.3 Reset Timing



Notes:

1. $\overline{\text{RESET}}$ transitions must be synchronized with BCLK rising edges.
2. CLIPPER bus is inactive until first instruction fetch.
3. Internal CPU startup time.
4. CPU diagnostics execution if $\overline{\text{URDIAG}}$ was asserted during the 2 BCLK cycles following release of $\overline{\text{RESET}}$.
5. Fetch from boot space.
6. $\overline{\text{URDIAG}}$ is asserted during $\overline{\text{RESET}}$ if CPU diagnostics execution prior to instruction execution is desired. $\overline{\text{URDIAG}}$ must remain asserted for at least 2 BCLK periods following release of $\overline{\text{RESET}}$ to assure recognition, and can then either remain asserted or be released with no further effect on CLIPPER Module operation.

Were the CPU to ignore these error conditions and continue execution, effects on the system could be catastrophic. A faulty or "lost" CPU could execute random writes to memory and I/O, for instance, corrupting data in both main memory and secondary storage.

The CLIPPER CPU offers protection from catastrophic failure by stopping program execution immediately upon detection of an unrecoverable fault condition, before the system is corrupted. It then asserts the Unrecoverable Fault signal ($\overline{\text{URF}}$) as a hardware indication that the CPU is halted due to an unrecoverable error, and that human intervention is required to correct the problem.

7.2.9 Reset and Diagnostics Control

The $\overline{\text{RESET}}$ signal is asserted to force the CLIPPER Module to a known initialized state. It is normally asserted when power is initially applied to the CLIPPER Module, remains asserted for sufficient time to ensure full initialization, then is released to force instruction execution from Boot space as shown in Figure 7.3.

The response of each CAMMU to assertion of $\overline{\text{RESET}}$ is as follows:

1. All LV (Line Valid) flags in the cache are cleared (set to 0).
2. All U (Used) flags in the cache are cleared.
3. All UV (User Valid) flags in the TLB are cleared.
4. All SV (Supervisor Valid) flags in the TLB are cleared.
5. All D (Dirty) flags in the TLB are cleared.
6. All R (Referenced) flags in the TLB are cleared
7. The Control Register is set to 3F.

The response of the CPU is as follows:

1. The PSW. The Trace Trap Enable (bit 23) is cleared. All other bits are unaffected by reset and should be initialized by software.
2. The SSW. Bits 8, 23, and 26 through 31 are cleared. No other bits are affected by reset. This register should be initialized by software after reset.

Assertion of reset therefore places the CLIPPER Module in unmapped supervisor mode with all traps and conditional interrupts disabled. In addition, all CLIPPER Bus active LOW signals are pulled HIGH (via pullup resistors), and all active HIGH signals are forced LOW. BCLK continues clocking normally.

The state of Apply Diagnostics ($\overline{\text{URDIAG}}$) during the two BCLK cycles following release of reset determines whether the CLIPPER Module CPU executes internal diagnostics before executing from boot code (see Figure 7.3). This is a powerful feature of the module which allows self test of major functions of the CPU without test equipment, and without chip removal. Failure during diagnostics is reported by assertion of the Unrecoverable Fault ($\overline{\text{URF}}$) CLIPPER Bus signal.

7.2.9.1 Self Test

The CLIPPER Module self test checks most major functions of the CPU, but not all. It is intended to be a first-level check of the CPU, which is in fact used to initially test individual CPU die during fabrication. The test executes approximately 700 instructions in about 4500 MCLK periods, using operands which test the CPU under worst-case conditions where possible. Worst-case carries, overflows, and sign extensions are tested, for example.

The following CPU operations and functions are tested:

- Pipeline resource management.
- Integer and Floating-Point execution units.
- General purpose register files.
- Integer bypass mechanism.
- Transition between supervisor and user modes.
- Temporary (hidden) registers.
- Macro branches.
- All addressing mode computations.
- Arithmetic shift, logical shift, and rotate instructions.
- Integer multiply and divide.
- Single- and double-precision floating-point instructions.
- Floating-point status bits.

CPU operations which require external response to instruction execution are not exhaustively tested. These include exception conditions, branches, loads, stores, pushes and pops, and I-CAMMU and D-CAMMU interfaces.

7.3 Bus Protocol

CLIPPER Bus operations are governed by the following rules:

1. A bus master cannot introduce wait states. This requires that a bus master be able to transfer data at the maximum rate allowed by the bus protocol.
2. Slaves may introduce wait states by delaying the assertion of RDYi (Ready) on the CLIPPER Bus. Wait states can be introduced between the address and data phases of an operation by delaying RDYi and between data words in a quadword transfer by toggling RDYi.
3. All CLIPPER Bus signals must be sampled on the positive edge of BCLK.
4. All signals must meet required setup and hold times with respect to the positive transition edge of BCLK. Signals must not transition within the Tsu (setup time) of BCLK rising edge; otherwise, undefined states within the CLIPPER Module can result.
5. If RDYo (Ready Out) is asserted by a CAMMU while CBSY is active, the memory interface must abort its data transfer. Data will be transferred instead by the CAMMU which asserts RDYo (see 7.4.1.1).

7.3.1 Bus Arbitration

A bus arbitration unit which arbitrates control of the CLIPPER Bus must be implemented in systems utilizing the CLIPPER Module. The unit must be capable of receiving bus requests from each of the possible bus masters via Bus Request lines BRx (where "x" identifies a bus request from a particular bus master), and must be able to assert a Bus Grant (BGx) for each bus master. The unit should support priority assignment such that in cases of multiple requests for the bus, the bus arbitration unit grants the bus to the highest-priority requesting device.

A bus master must hold BRx asserted during its entire access of the CLIPPER Bus, then should release BRx as soon as possible after completion of its bus operation in order to maintain high system throughput. The bus arbitration unit should hold BGx asserted until after the bus master has released BRx, and should release BGx no later than one BCLK after release of BRx.

7.3.2 Bus Error

Bus operations are aborted by the assertion of $\overline{\text{BERR}}$ (Bus Error). Bus error conditions must be detected by user-designed logic implemented in the CLIPPER system, which then asserts $\overline{\text{BERR}}$ and an interrupt request (via the interrupt logic). The CAMMU terminates the system bus access and releases the bus when it detects the assertion of $\overline{\text{BERR}}$. The CPU should use the interrupt request to vector to a routine designed for resolution of the bus error condition. The state of the CLIPPER Bus associated with the bus error may be stored by the bus error logic for use by the bus error interrupt service routine. Timing for bus error aborts is shown in Figure 7.22.

7.3.3 Wait States

Slow devices can introduce wait states by delaying assertion of RDY on the CLIPPER Bus during bus operations. Wait states consist of an integral number of BCLK periods during which time the bus master remains in a "waiting state" until the

Figure 7.4 Single Word Read (No Wait States)

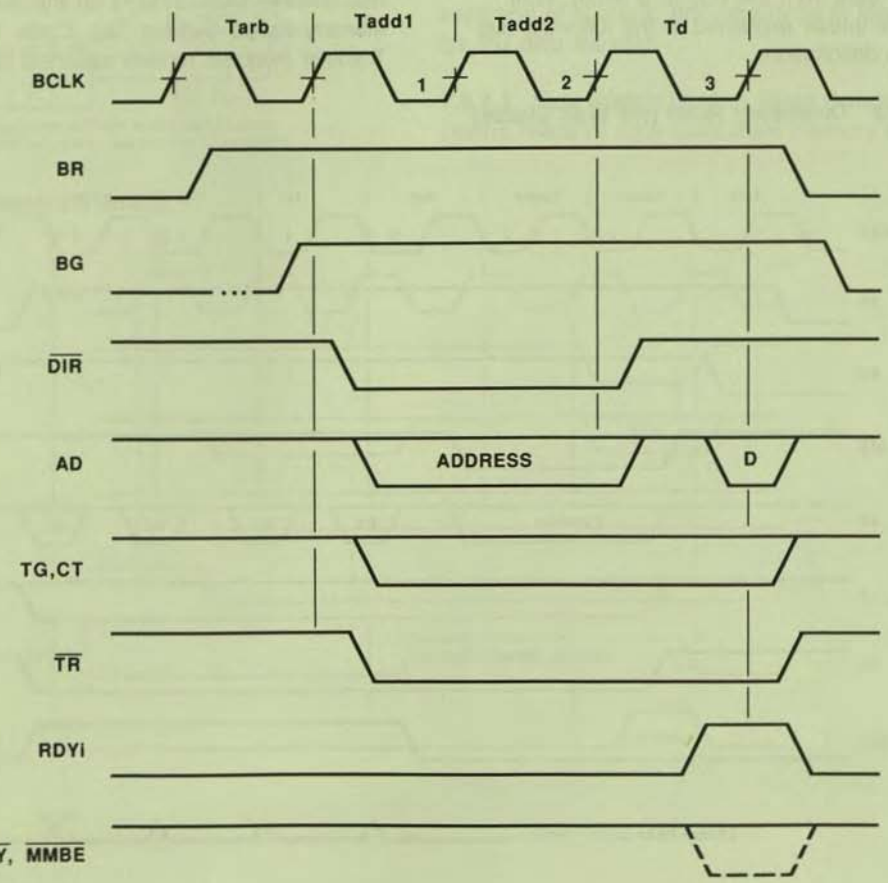
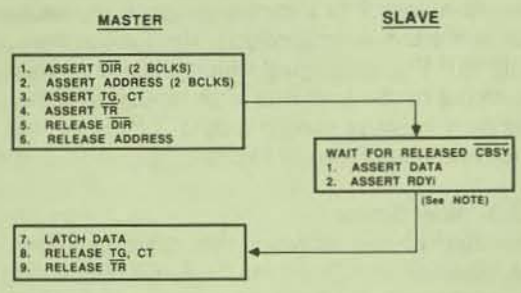


Figure 7.5 Single Word Read Flowchart



NOTE: Slave aborts data assertion if RDYi is asserted while CBSY is asserted (see Bus Watch Description).

slave device asserts RDYi to indicate that it has asserted data on the bus (if a read operation), or has read data from the bus (if a write). Wait states are further explained in the following bus operation descriptions.

7.4 CLIPPER Bus Operations

A CLIPPER Bus operation begins when a bus master requests the bus by asserting its Bus Request signal, then receives Bus Grant from the bus arbitration unit. The bus master then proceeds with the bus operation while holding its Bus Request asserted, thereby retaining mastership of the bus. The bus master releases its Bus Request upon completion of the operation, allowing other bus masters to gain control of the bus. Timing and flowcharts associated with various bus operations are shown in Figures 7.4 through 7.22.

7.4.1 Read Operation

Upon receiving Bus Grant, the master (possibly a CAMMU) asserts Direction Control (DIR) Transfer Request (TR), the Memory Space System Tag (TG<2:0>), Cycle Type (CT<5:0>), and a real address (AD<31:0>) on the bus. The Memory Space System Tag, Cycle Type, and Transfer Request remain asserted during the

Figure 7.6 Quadword Read (No Wait States)

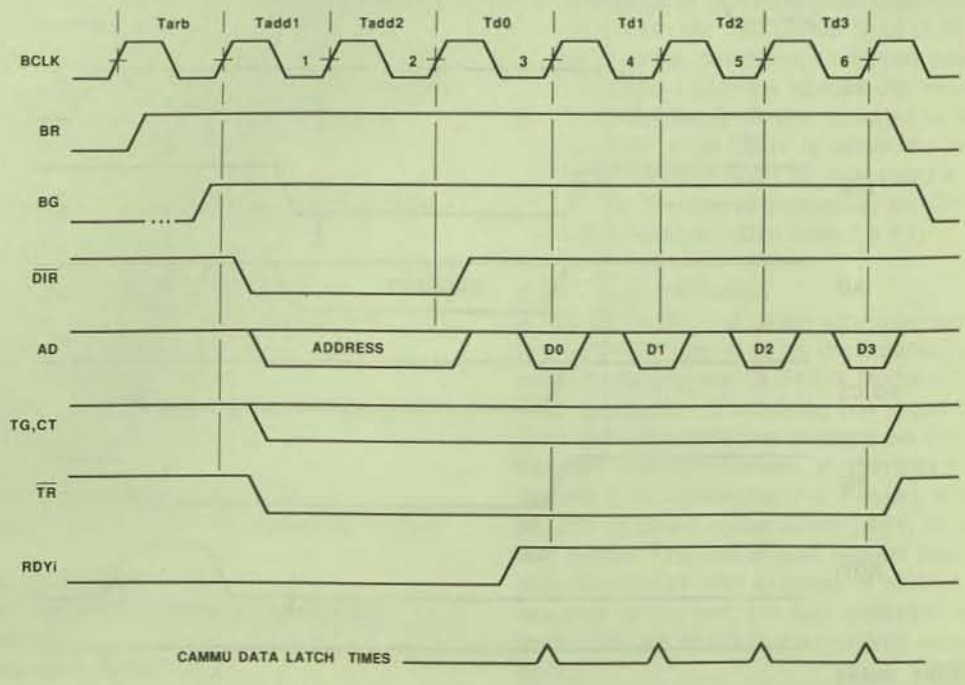
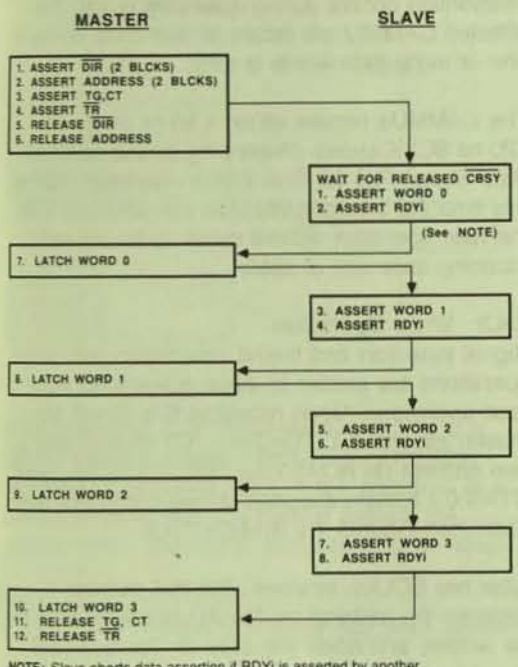


Figure 7.7 Quadword Read Flowchart



NOTE: Slave aborts data assertion if RDYi is asserted by another device while CBSY is asserted (see Bus Watch Description)

entire operation. However, the master three-states the multiplexed address/data lines AD<31:0> after two BCLK cycles to make the lines available for data transfer by the slave device and releases DIR (see Figures 7.4 through 7.7).

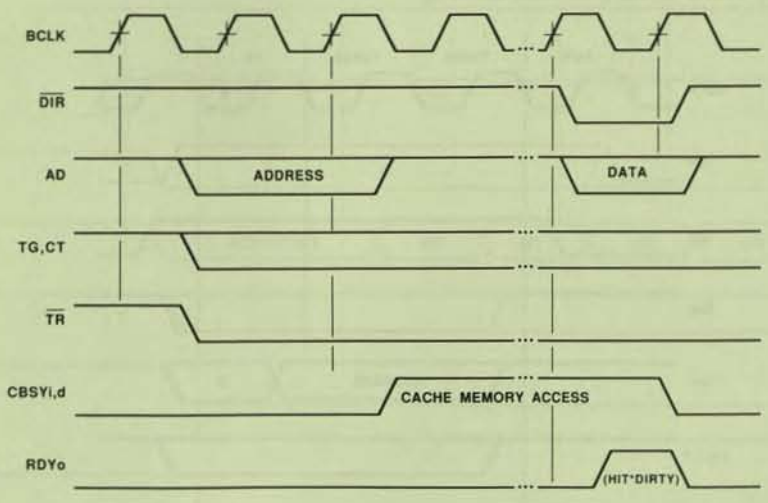
The bus master then waits for the slave device to assert RDYi, indicating that the data is on the bus, and latches the data on the same positive transition of BCLK that it detects RDYi. The slave device can respond with data immediately after the address/data lines are three-stated by the bus master, or can respond later as required by delaying assertion of RDYi thereby introducing wait states.

The minimum number of BCLK cycles required for a read operation is three, excluding bus arbitration requirements: two cycles are required for assertion of address, and one cycle is required for the data transfer.

7.4.1.1 Bus Watch During Read Operations

During reads of copy-back main memory space

Figure 7.8 Watch I/O Reads



NOTE:
I/O device reads from main memory (e.g. main memory - disk).

by an I/O bus master (as indicated by the TG and CT lines), each CAMMU with Watch I/O Reads mode enabled asserts $\overline{\text{CBSY}}$, indicating that it is checking whether the main memory data to be read is current with cached data (see Figure 7.8). If the data is cached and the cached data is dirty (the main memory data is not current), the affected CAMMU asserts the current cached data onto the AD bus, then releases $\overline{\text{CBSY}}$ and asserts RDYo. The I/O master must then latch the data on the positive BCLK transition following assertion of RDYo. If the data is not cached or the cached data is not dirty, the affected CAMMU releases $\overline{\text{CBSY}}$, allowing transfer of main memory data to the bus master.

The main memory interface must monitor CBSY and RDYo and allow main memory data response only if $\overline{\text{CBSY}}$ is not asserted, indicating that there will be no CAMMU intervention resulting from CAMMU Bus Watch. If RDYo is asserted by the CAMMU, the memory interface must abort its operation because the CAMMU is responding with more recent cache data.

This Bus Watch sequence applies to both single-word reads and quadword reads. If Bus Watch intervention occurs during quadword reads, the affected CAMMU will return all four data words if one or more data words is dirty.

The CAMMUs require either 4 60 ns or 2 120 ns BCLK cycles (depending on the state of Rate Signal Pin) for Bus Watch checking. During this time the memory interface can proceed with the read operation without delay up to but not including assertion of data.

7.4.2 Write Operation

Signal assertion and timing associated with write operations are similar to those associated with read operations. Upon receiving Bus Grant the master asserts $\overline{\text{TR}}$, TG<2:0>, CT<5:0>, and a real address on AD<31:0>. $\overline{\text{TR}}$, TG<2:0>, and CT<5:0> remain asserted during the entire operation (see Figures 7.9 through 7.13).

After two BCLKs, however, the bus master replaces the address on the AD bus with data to be written, and holds the data on the bus until the slave acknowledges receiving the data by asserting RDYi.

Figure 7.9 Single Word Write (No Wait States)

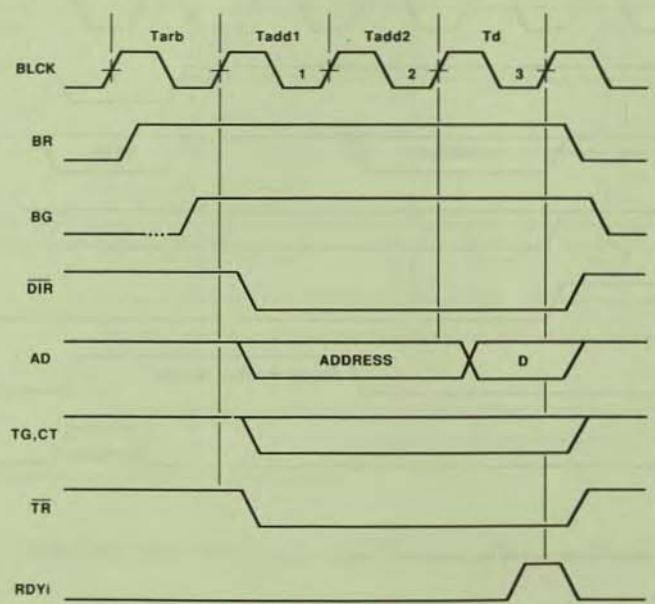
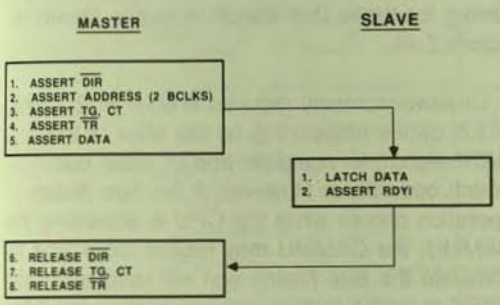


Figure 7.10 Single Word Write Flowchart



If the operation is a single-word write, the bus master releases the bus immediately folowing detection of asserted RDYi. If the operation is a quadword write, the bus master asserts the second, third, and fourth data words upon detection of successive RDYi assertions. The slave can thus introduce wait states for each word transfer of the quadword write.

The minimum number of BCLK cycles required for a single word write operation is three, excluding bus arbitration requirements. Two cycles are required for assertion of address, and one cycle is required for the data transfer.

7.4.2.1 Bus Watch During Write Operations

A CAMMU invokes Watch I/O Writes, if enabled by the Control Register, when an I/O device writes to cacheable main memory pages; and invokes Watch CPU Writes, if enabled, when a CPU (via a D-CAMMU) writes to cacheable, shared main memory pages. Both Bus watch modes function identically. If the write operation invoking one of these modes is a single-word write operation, each affected CAMMU updates its cache with the data written to main memory.

Figure 7.11 Quadword Write (No Wait States)

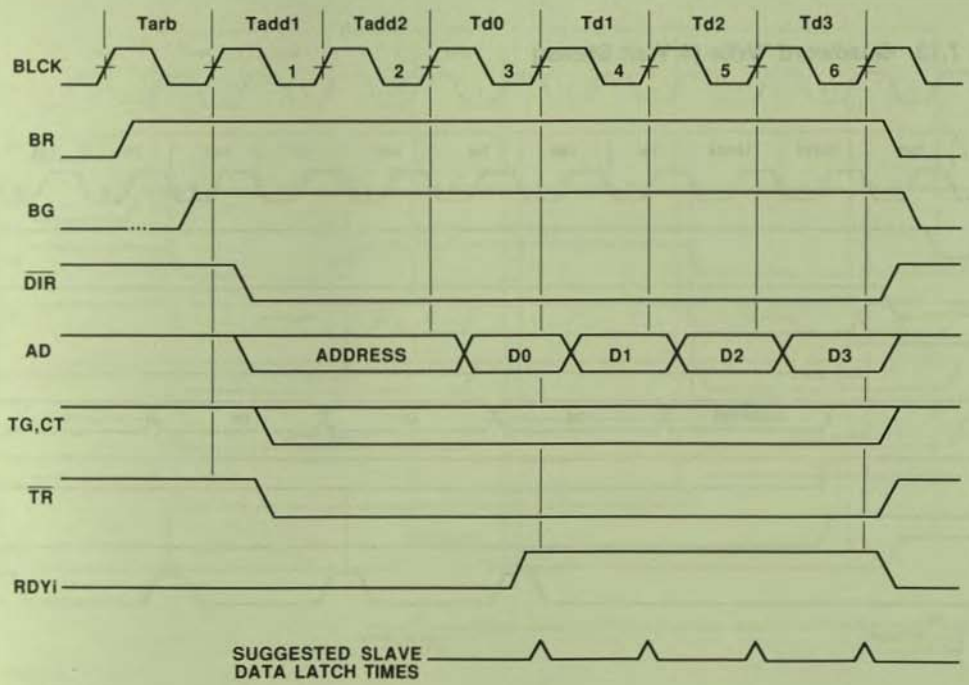


Figure 7.12 Quadword Write Flowchart

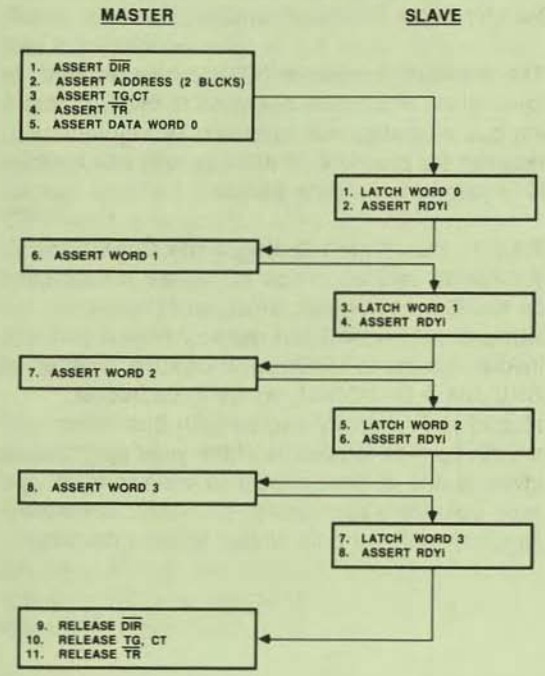
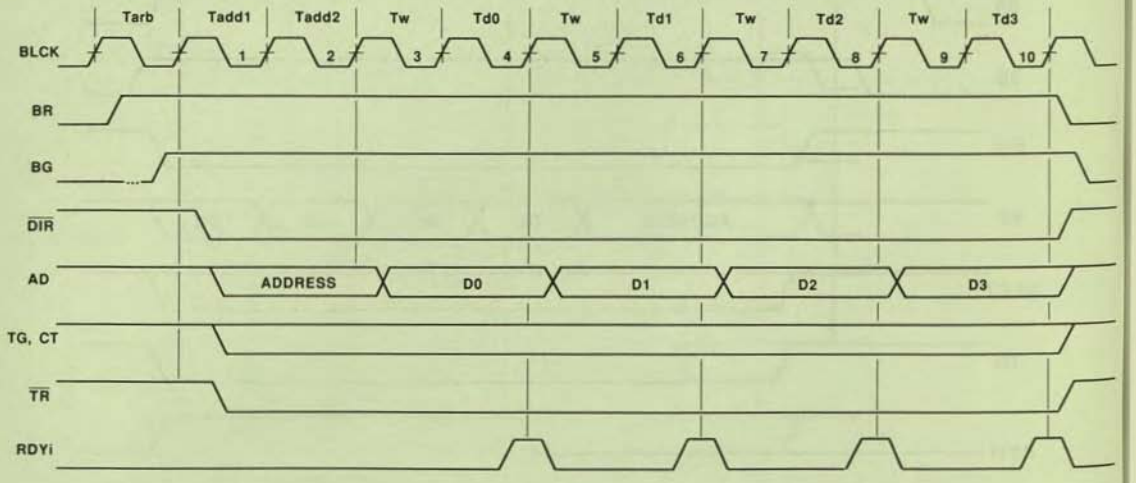


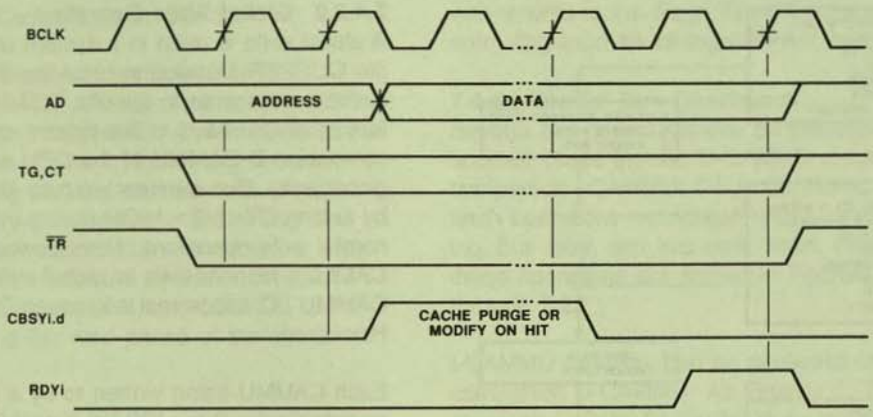
Figure 7.13 Quadword Write (4 Wait States)



If the write operation is a quadword write, each affected CAMMU invalidates the cache line containing the data addressed in main memory. Timing for these Bus Watch modes is shown in Figure 7.14.

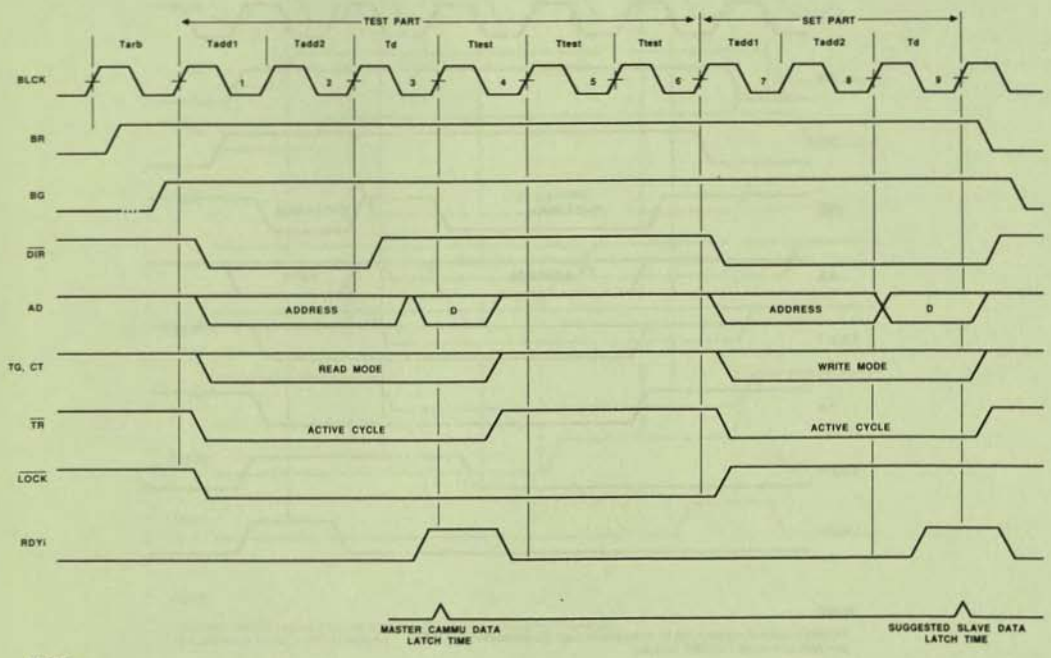
A CAMMU normally requires 4 60ns or 2 120ns BCLK cycles (depending on the state of the RATE signal) to complete one of these Bus Watch operations. However, if the Bus Watch operation occurs while the CPU is accessing the CAMMU, the CAMMU may require more time to complete the Bus Watch and will keep asserting CBSY to inhibit further bus operations until it has completed the task. The bus master can complete a quadword write operation while the CAMMU executes its Bus Watch operation because the CAMMU invalidates a cache line that is hit. If the operation is a single word write, the memory interface must not assert RDYi until CBSY is released, to allow the CAMMU access to the data for cache update if the cache is hit. If the CPU is accessing the cache line pointed to by the bus address, the CPU will complete its

Figure 7.14 Watch CPU and I/O Writes



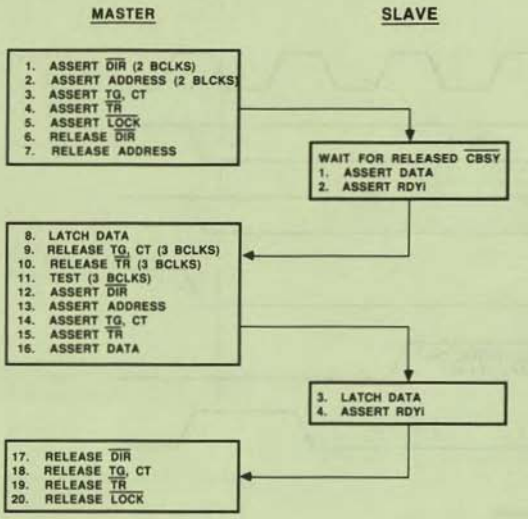
NOTES:
1. I/O WRITE - I/O device writes into main memory (e.g., disk → main memory).
2. CPU WRITE - One of the CAMMUs writes into a shared area of main memory.
3. RDYi is asserted by the memory interface.

Figure 7.15 Read-Modify-Write (Test and Set)



NOTE:
Test - CAMMU determines if the set part of the operation is needed.

Figure 7.16 Read-Modify-Write Flowchart



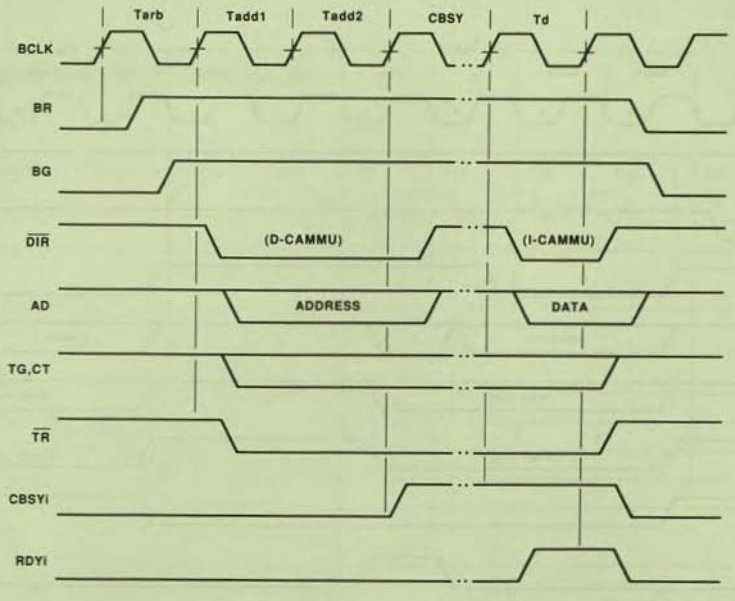
access before the CAMMU modifies or invalidates the line.

7.4.2.2 Global Write Operation

A global write is used in a system utilizing multiple CLIPPER Modules to reset the TLBs or caches, or to write to specific TLB lines or registers in all CAMMUs in the system except the companion D-CAMMU of the CPU executing the global write. Bus masters execute global writes by setting CT<3:2> HIGH during the otherwise normal write operations. Note, however, that CAMMUs respond only to global writes to CAMMU I/O space real addresses Cnn and Dnn Hex.

Each CAMMU being written to by a global write asserts Cache Busy (CBSYi if an I-CAMMU, CBSYd if a D-CAMMU) during the write to inhibit

Figure 7.17 D-CAMMU Read from Companion I-CAMMU



NOTE:

I-CAMMU internal registers can be accessed through the companion D-CAMMU (D-CAMMU of same CLIPPER module).

further bus activity until it has completed internal tasks associated with the write. System logic is required which detects global writes and asserts Ready when Cache Busy is released.

7.4.3 Read-Modify-Write Operation

A read-modify-write operation is a combination of a read operation, followed by a write operation (see Figures 7.15 and 7.16). Timing and protocol associated with the read and the write phases of the read-modify-write operation are similar to those for single reads and writes. However, Bus Request must be asserted by the bus master during the entire operation, and $\overline{\text{LOCK}}$ is asserted during the read phase of the operation.

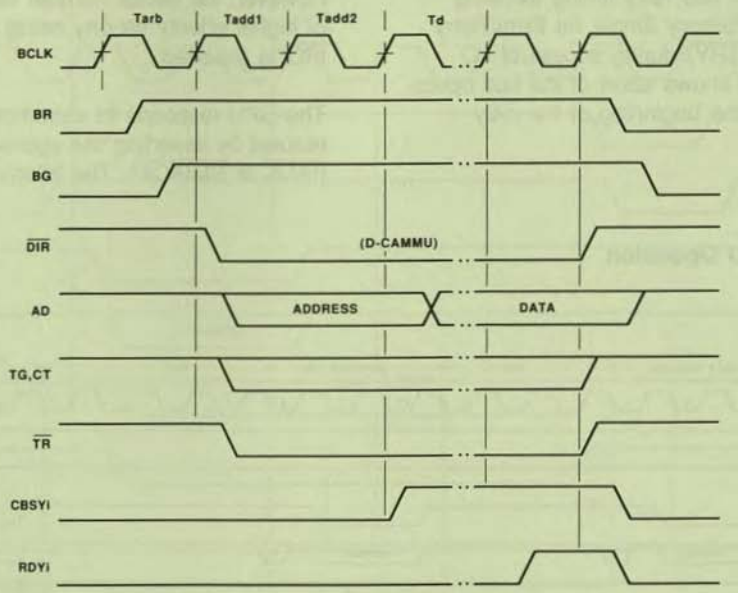
Read-modify-write operations are executed during test-and-set instruction executions. They are also executed during address translation when the DTU accesses main memory twice to read the Page Table Directory and Page Table and follows with a write to the Page Table if a Page Table entry flag must be changed.

7.4.4 Special Bus Operations

Several bus operations can be classified as special. These include D-CAMMU access of its companion I-CAMMU, DTU operations, CAMMU fetch operations associated with copyback caching, bus retry, and bus error abort. Examples of these operations are shown in Figures 7.17 through 7.22.

I-CAMMU registers can be accessed only by the companion D-CAMMU. As Figures 7.17 and 7.18 show, the I-CAMMU Cache Busy and Ready signals control the data transfer. The I-CAMMU

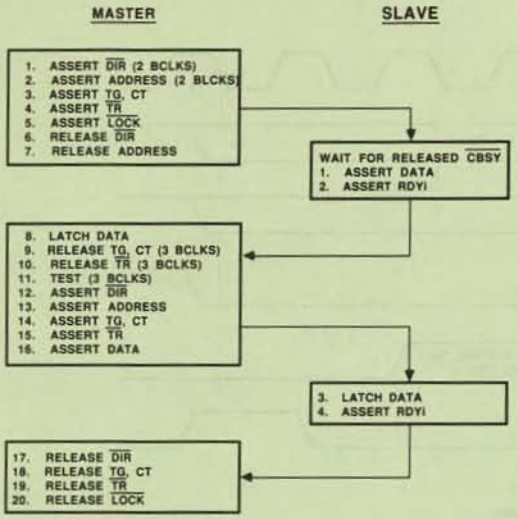
Figure 7.18 D-CAMMU Write to Companion I-CAMMU



NOTE:

I-CAMMU internal registers can be accessed through the companion D-CAMMU (D-CAMMU of same CLIPPER module).

Figure 7.16 Read-Modify-Write Flowchart



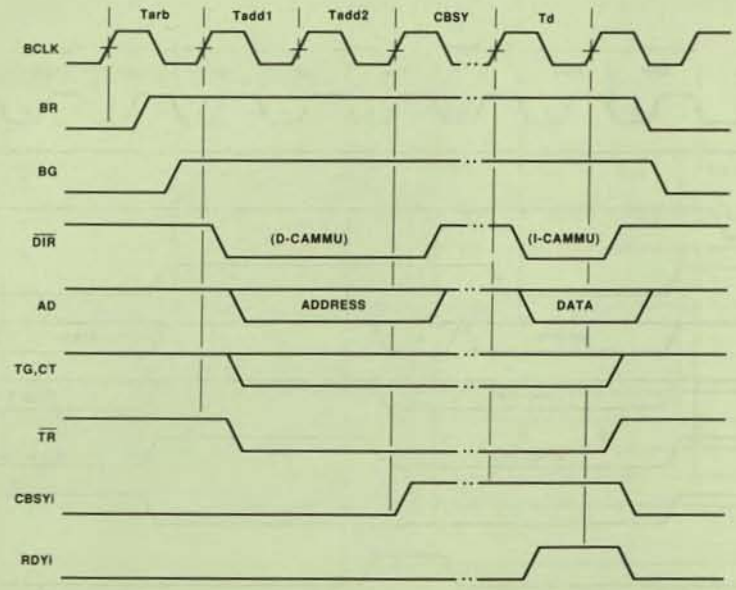
access before the CAMMU modifies or invalidates the line.

7.4.2.2 Global Write Operation

A global write is used in a system utilizing multiple CLIPPER Modules to reset the TLBs or caches, or to write to specific TLB lines or registers in all CAMMUs in the system except the companion D-CAMMU of the CPU executing the global write. Bus masters execute global writes by setting CT<3:2> HIGH during the otherwise normal write operations. Note, however, that CAMMUs respond only to global writes to CAMMU I/O space real addresses Cnn and Dnn Hex.

Each CAMMU being written to by a global write asserts Cache Busy (CBSYi if an I-CAMMU, CBSYd if a D-CAMMU) during the write to inhibit

Figure 7.17 D-CAMMU Read from Companion I-CAMMU



NOTE:

I-CAMMU internal registers can be accessed through the companion D-CAMMU (D-CAMMU of same CLIPPER module).

further bus activity until it has completed internal tasks associated with the write. System logic is required which detects global writes and asserts Ready when Cache Busy is released.

7.4.3 Read-Modify-Write Operation

A read-modify-write operation is a combination of a read operation, followed by a write operation (see Figures 7.15 and 7.16). Timing and protocol associated with the read and the write phases of the read-modify-write operation are similar to those for single reads and writes. However, Bus Request must be asserted by the bus master during the entire operation, and $\overline{\text{LOCK}}$ is asserted during the read phase of the operation.

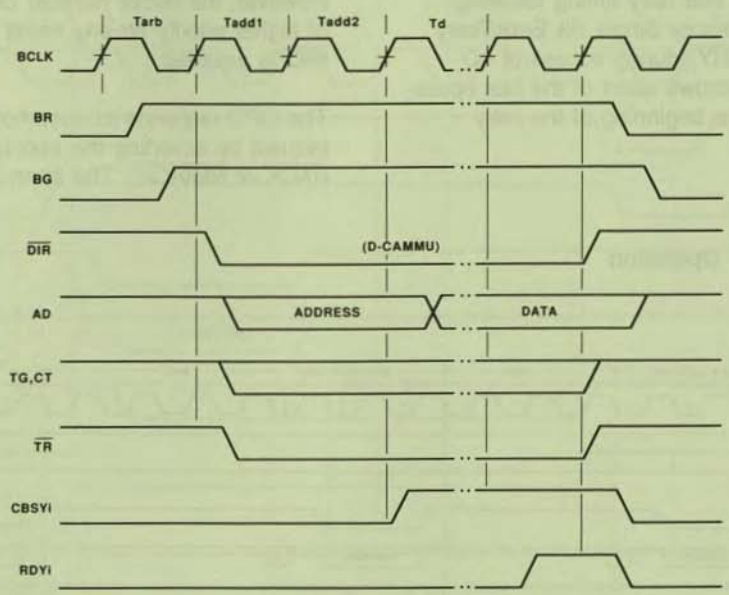
Read-modify-write operations are executed during test-and-set instruction executions. They are also executed during address translation when the DTU accesses main memory twice to read the Page Table Directory and Page Table and follows with a write to the Page Table if a Page Table entry flag must be changed.

7.4.4 Special Bus Operations

Several bus operations can be classified as special. These include D-CAMMU access of its companion I-CAMMU, DTU operations, CAMMU fetch operations associated with copyback caching, bus retry, and bus error abort. Examples of these operations are shown in Figures 7.17 through 7.22.

I-CAMMU registers can be accessed only by the companion D-CAMMU. As Figures 7.17 and 7.18 show, the I-CAMMU Cache Busy and Ready signals control the data transfer. The I-CAMMU

Figure 7.18 D-CAMMU Write to Companion I-CAMMU



NOTE:

I-CAMMU internal registers can be accessed through the companion D-CAMMU (D-CAMMU of same CLIPPER module).

asserts its Cache Busy upon being addressed, and holds it asserted until it has completed current operations and is ready for the data transfer. The I-CAMMU asserts Ready (RDYoi) for one BCLK cycle, then releases Cache Busy and Ready, resulting in termination of the operation.

A DTU operation consists of a read of a Page Table Directory followed by a read of a Page Table, followed, if required, by a write to the accessed Page Table to modify a Referenced or Dirty Flag. This is an integral operation indicated by the assertion of $\overline{LOCK}$ as shown in Figure 7.19.

Timing for cache line replacement resulting from a cache miss is shown in Figure 7.20. As the figure shows, the operation consists of a quadword write of the cache line to memory if the line is dirty, followed by a quadword read of the replacement line into the cache.

Figure 7.21 shows bus retry timing following assertion of the Memory Single Bit Error/Retry signal ($\overline{MSBE}/\overline{RETRY}$) during access of I/O space. The figure shows abort of the bus operation, followed by the beginning of the retry operation.

Bus operation abort timing in response to assertion of Bus Error is shown in Figure 7.22.

7.4.5 Interrupts

$\overline{IVEC}$<7:0> is an interrupt bus within the CLIPPER Bus which is used only to transfer interrupt vector numbers from the system directly to the CLIPPER CPU, as already explained. $\overline{NMI}$ is the nonmaskable interrupt request signal, and $\overline{IRQ}$ is the maskable interrupt request signal which requires that an associated vector number be transferred on the interrupt bus.

An interrupt controller must be implemented in a CLIPPER system which supports more than one $\overline{IRQ}$ interrupt. In cases of multiple $\overline{IRQ}$ interrupt requests, it must select the interrupt with the highest priority by asserting the vector number associated with the interrupt on $\overline{IVEC}$<7:0>. The interrupt controller must assert each interrupt request and the associated vector number together on the same positive edge of BCLK. However, the vector number can change to one of higher priority on any rising BCLK edge while $\overline{IRQ}$ is asserted.

The CPU responds to assertion of an interrupt request by asserting the appropriate acknowledge ($\overline{IACK}$ or $\overline{NMIACK}$). The interrupt controller can

Figure 7.19 DTU Operation

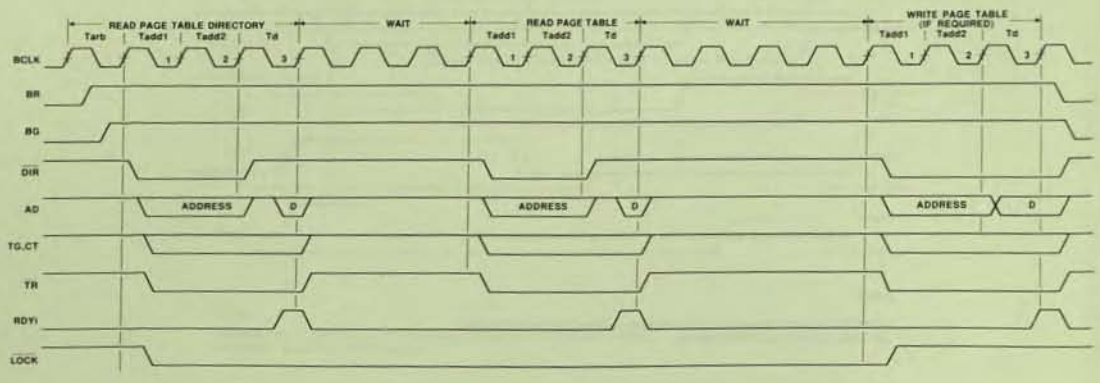


Figure 7.20 Cache Line Replacement

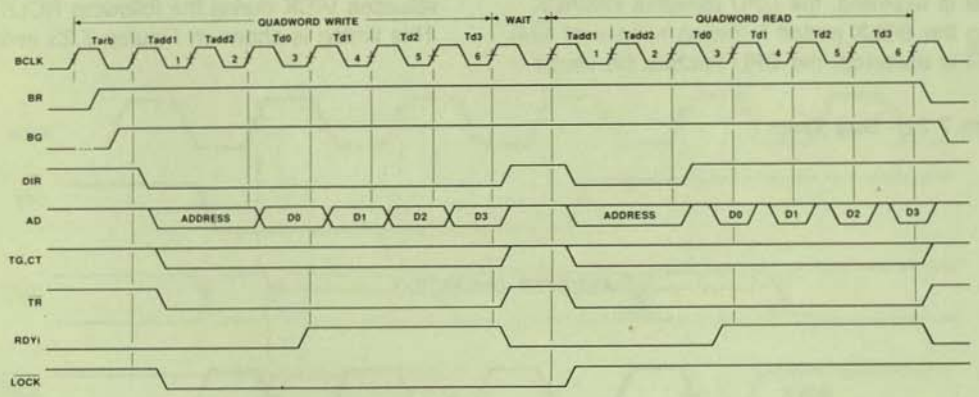
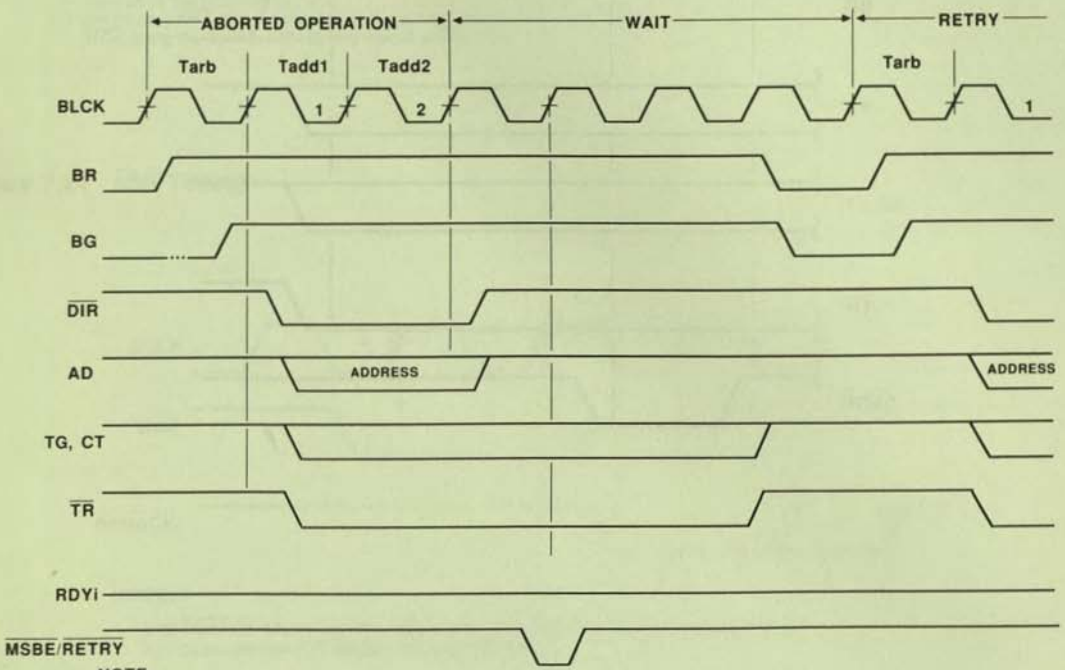


Figure 7.21 Bus Retry (Single Word Read Example)



NOTE:
Retry occurs when $\overline{\text{MSBE}}/\overline{\text{RETRY}}$ is asserted while RDYi is released during access of I/O space only (TG = 4).

then release the interrupt request.

If $\overline{NMI}$ is asserted, the CPU releases $\overline{NMIACK}$ during the BCLK period following release of $\overline{NMI}$. If $\overline{IRQ}$ is asserted, the CPU latches the vector number present on the $\overline{IVEC}$ bus when it detects release of $\overline{IRQ}$ on a rising edge of BCLK, then releases $\overline{IACK}$ during the following BCLK period. This timing is shown in Figures 7.23 and 7.24.

Figure 7.22 Bus Error

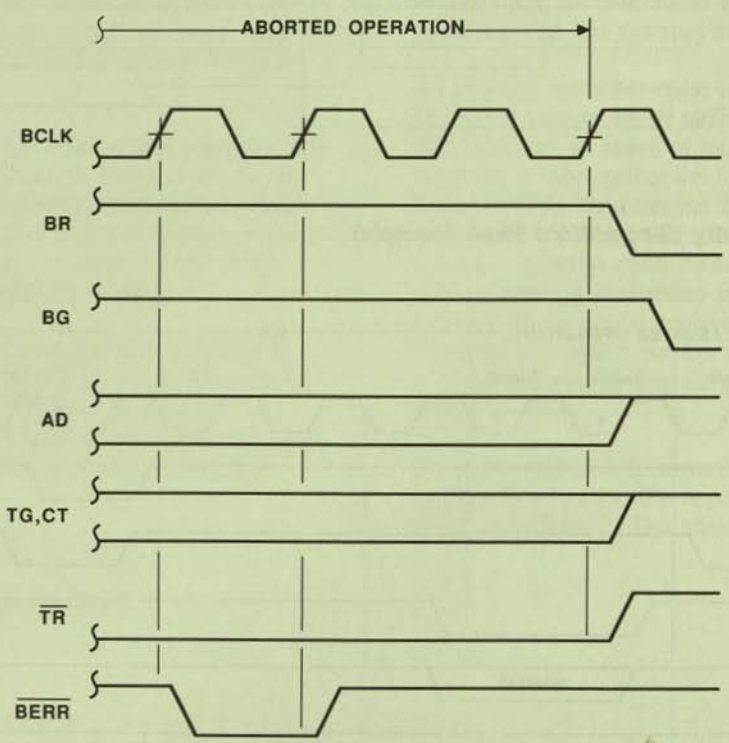
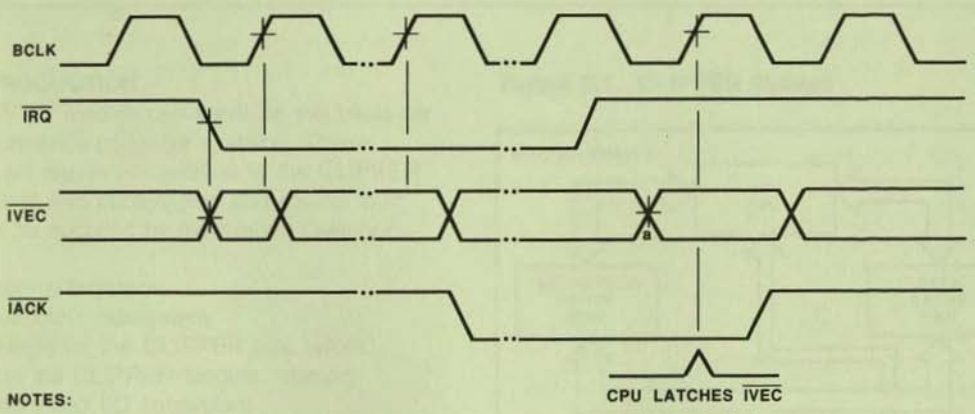


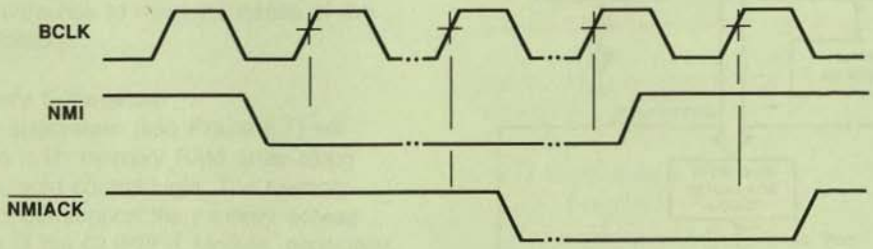
Figure 7.23 $\overline{IRQ}$ Timing



NOTES:

1. After $\overline{IACK}$ returns high the $\overline{IRQ}$ line must be high for one clock before another interrupt request returns $\overline{IRQ}$ low.
2. The IVEC lines must only change to a higher priority when $\overline{IRQ}$ is low. The higher priority number must be on the IVEC lines by "a".
3. CPU latches IVEC on the rising edge of BLCK following release of $\overline{IRQ}$. The CPU releases $\overline{IACK}$ during the BLCK period following release of $\overline{IRQ}$.

Figure 7.24 $\overline{NMI}$ Timing



NOTE:

1. After $\overline{NMIACK}$ returns high the $\overline{NMI}$ line must be high for one clock before another non-maskable interrupt request returns $\overline{NMI}$ low.

Chapter 8
Hardware Design Considerations

8.1 INTRODUCTION

The CLIPPER Module can serve as the basis for high-performance computer systems. These systems will require, in addition to the CLIPPER Module itself, two subsystems and control logic that must be supplied by the system designer:

- a memory subsystem
- an input/output subsystem
- control logic for the CLIPPER bus, which connects the CLIPPER Module, memory subsystem and I/O subsystem

This chapter addresses design issues associated with each of these subsystems. Discussions are general, in the belief that designers will benefit most from descriptions of the overall CLIPPER system architecture and of key system protocol requirements. Typical examples of how these subsystems can be implemented are described, using high-level block diagrams. The examples are simple, straightforward approaches intended to outline the principles used when interfacing to the CLIPPER Bus and are by no means exhaustive. The CLIPPER Bus supports a wide variety of design approaches to meet the needs of the desired application.

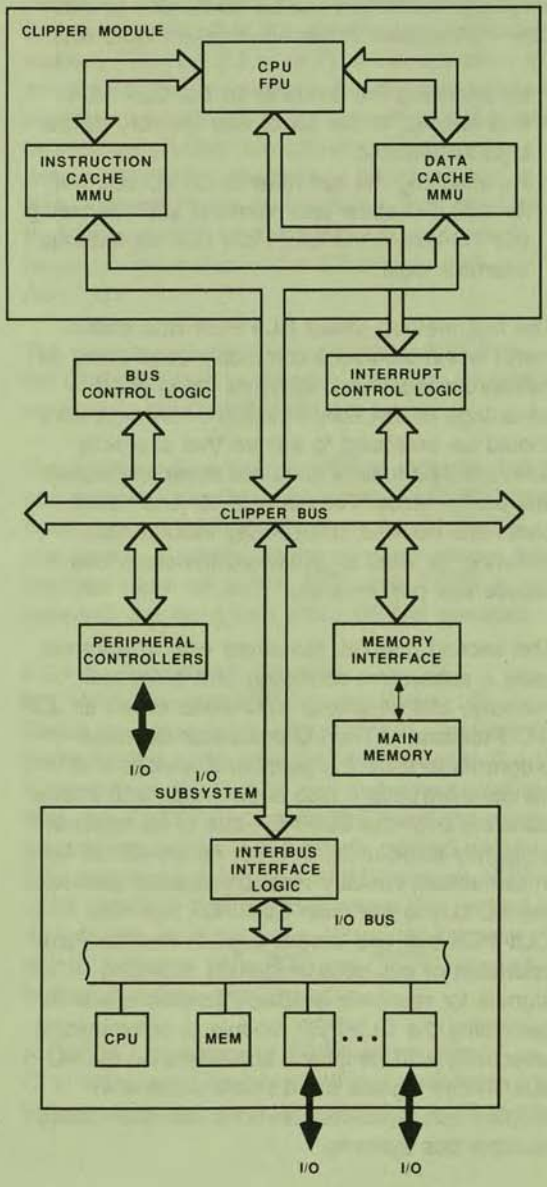
8.1.1 Memory Subsystem

The memory subsystem (see Figure 8.1) will consist of the main memory RAM array along with its associated control logic. The memory control logic must support the memory access requirements of the CLIPPER Module, especially:

- quadword data transfers using burst-mode timing
- bus watch

In addition, the memory subsystem will usually support more conventional capabilities, such as error detection and correction.

Figure 8.1 CLIPPER System



8.1.2 I/O Subsystem

The I/O facilities of a CLIPPER-based computer system are provided by peripheral devices — disks, terminals, communication links, or printers — that interact directly with the outside world. These peripheral devices interface to the system via *peripheral controllers*, control circuitry specific to each device, for example, disk controllers, serial communication controllers, etc. These peripheral controllers can be connected to the CLIPPER system in two ways (see Figure 8.1):

- By attaching the controller to the CLIPPER Bus directly, in the same way memory control logic is attached.
- By attaching the controller to an I/O bus with its own processor and memory, and interfacing this I/O bus to the CLIPPER Bus via interbus interface logic.

The first method (direct CLIPPER Bus attachment) is the technique commonly used today for microprocessor-based systems. Its principal advantage is simplicity of design. However, care should be exercised to ensure that a directly connected peripheral does not adversely impact bus performance. For example, 8- and 16-bit peripheral devices, unless they include data buffering for word or quadword transfers, can reduce bus performance.

The second method, pioneered with mainframes, uses a subsystem consisting of a processor, memory, and peripheral controllers, called an *IOP* (I/O Processor). The I/O processor executes programs to drive the peripheral devices and handle interrupts; it also buffers data and allows transfers over the CLIPPER bus to be made in efficiently sized units. In order for an IOP to be implemented, circuitry must be used to connect the I/O bus to the main CLIPPER bus. The CLIPPER bus has several signals that facilitate operation of bus adapter circuits, including signals for resolving deadlock conditions and for permitting the CLIPPER module to communicate effectively with peripheral controllers on the I/O bus. These signals are discussed below in Section 8.3. Systems with IOPs are often called *multiple bus* systems.

8.1.3 CLIPPER Bus Control and Interrupt Logic

Besides memory and peripherals, a CLIPPER-based system will need special control logic for the CLIPPER Bus itself. In particular, logic must be provided for *interrupt control* and *bus arbitration*. The interrupt controller is responsible for informing the processor that a peripheral device is requesting service and passing a vector number over the CLIPPER's Interrupt Vector bus that identifies the interrupt. A bus arbiter is used to resolve bus access conflicts between bus masters, that is, devices that are capable of initiating a transfer over the CLIPPER bus.

8.2 MEMORY SUBSYSTEM

The main memory requires an interface to the CLIPPER Bus. The interface controls all data transfers between the bus and main memory, and may contain parity error detection or memory error detection and correction logic. The interface must support CLIPPER Module Bus Watch if caching is used, and may be designed to support Clear Dirty operation (see Section 8.2.1.2).

The function of the main memory interface is best defined by the interface input and output signals. These signals are defined from the memory interface point of view as follows:

INPUT SIGNALS

TG<2:0>: These Memory Space System Tag signals identify bus operations as main memory accesses.

CT<5:0>: These Cycle Type signals identify operations as reads or writes, identify byte/halfword/word and quadword transfers, and identify byte and halfword boundaries.

$\overline{TR}$: This signal indicates, when active, that a bus operation is in progress.

AD<31:0>: These bus signals transfer address and data from the CLIPPER Module to the memory interface.

$\overline{CBSY}$: This signal is asserted to indicate that a CAMMU is busy executing Bus Watch. If $\overline{CBSY}$ is asserted during a read of main memory, the memory interface must not assert memory data on the AD bus. This allows the CAMMU to abort assertion of data by the memory interface, and to provide cached data (see $\overline{RDY}$).

$\overline{RDY}$: This bidirectional Ready signal is monitored by the memory interface to abort read operations. If $\overline{RDY}$ is asserted while $\overline{CBSY}$ is asserted, the memory interface aborts the operation.

BCLK: This system clock is used by the memory interface for synchronization.

OUTPUT SIGNALS

AD<31:0>: The memory interface transfers memory data to the CLIPPER Bus via these multiplexed Address/Data lines.

$\overline{RDY}$: The memory interface asserts $\overline{RDY}$ to indicate presence of memory data on the AD bus during read operations, and latching of data during write operations.

$\overline{MSBE}$/ The memory interface error detec-

$\overline{RETRY}$: tion logic asserts the Memory Single Bit Error signal to indicate detection of a corrected single-bit memory error.

$\overline{MMBE}$: The memory interface error detection logic asserts the Memory Multiple Bit Error signal to indicate detection of an uncorrected memory error.

These signals are fully described in Chapter 7. Timing associated with the signals is shown throughout Chapter 7, and timing parameters for the signals are provided in the *CLIPPER 32-Bit Microprocessor Product Description.*

8.2.1 Memory Interface Operation

Memory interface operation begins when $\overline{TR}$ is asserted, indicating that a bus operation is in progress. The memory interface uses TG<2:0> to determine whether the operation involves main memory (TG=0,1,2,3,6, or 7). If the operation is an access of main memory, the memory interface transfers the address to the memory DRAMs and determines whether the operation is a read or write, and whether the data to be transferred is a byte, halfword, word, or quadword. The interface transfers memory words, halfwords, and bytes based on the states of the CT signals and AD<1:0>.

The memory interface then transfers data from the CLIPPER Bus to memory if a write, or from memory to the CLIPPER Bus if a read.

The interface then asserts $\overline{RDY}$ to indicate that it has asserted memory data on the CLIPPER Bus (read), or has latched data from the CLIPPER Bus (write). However, during read operations the interface must not assert $\overline{RDY}$ while $\overline{CBSY}$ is asserted, but must wait until $\overline{CBSY}$ is released.

8.2.1.1 Bus Watch Support

A CAMMU asserts $\overline{CBSY}$ during Watch I/O Reads operation to indicate that it is checking its cache for the addressed data (see Section 7.4.1). If the data is not cached or is not dirty copy-back data, the CAMMU releases $\overline{CBSY}$ and the read proceeds normally. If the data is cached and is dirty copy-back data, the CAMMU asserts $\overline{RDY}$ while asserting $\overline{CBSY}$ and provides the cached data for the read operation. When the memory interface detects both $\overline{RDY}$ and $\overline{CBSY}$ asserted, it must abort the read operation.

A CAMMU also asserts $\overline{CBSY}$ during Watch CPU Writes and Watch I/O Writes operations to indicate that it is checking whether it must update

its cache with the data (single word, halfword, or byte write), or must invalidate a cache line (quadword write). If $\overline{CBSY}$ is asserted during a single word, halfword, or byte write, the memory interface must delay asserting of $\overline{RDY}$ until $\overline{CBSY}$ is released to ensure that the data remains on the AD bus until the CAMMU has had a chance to read it. The memory interface can proceed with quadword writes while $\overline{CBSY}$ is asserted if the bus arbiter monitors $\overline{CBSY}$ and does not grant the bus to a new master until $\overline{CBSY}$ is released. In all cases, masters executing multiple transfers requiring only one arbitration of the bus must wait for $\overline{CBSY}$ to release before beginning each transfer.

The memory interface must therefore monitor $\overline{CBSY}$ only during read operations by I/O (indicated by CT<5>=1) and during cachable single-word write operations (indicated by CT<4:2>=0, TG<2:0>=0-3) , and can ignore assertion of $\overline{CBSY}$ during other operations.

8.2.1.2 Clear Dirty Support

The memory interface can be designed to update the addressed memory location with the data supplied by a CAMMU as a result of Watch I/O Reads operation. Thus during an I/O read of data that is cached and dirty (most current data), memory can be updatad with the cache data at the same time that the I/O is reading the data. If this feature is designed into the memory interface, the Clear Dirty bit in the CAMMU Control Register can be set, forcing the Line Dirty bit of a hit line to clear as a result of Watch I/O Reads operation. This eliminates the need to write the line to memory when the page is replaced, unless the line becomes dirty again.

The CAMMU asserts $\overline{RDY}$ (RDYo) to indicate that it has asserted the cached data on the bus. The memory interface must transfer the data on the rising edge of BCLK during which $\overline{CBSY}$ and $\overline{RDY}$ are asserted. The memory interface cannot introduce wait states during this operation.

8.2.1.3 Memory Errors

The memory interface can incorporate error correction logic or parity logic. Error detection logic can be designed into the interface which detects corrected single bit data errors, and uncorrected errors. This logic should assert $\overline{MSBE/RETRY}$ when it detects a corrected error, and $\overline{MMBE}$ when it detects an uncorrected error. Parity errors must be regarded as uncorrected errors.

8.2.2 Boot Memory

CLIPPER system booting is preceeded by assertion of $\overline{RESET}$ for at least 100 BCLK cycles after VDD reaches the specified VDD minimum (see Figure 7.3). The CLIPPER Module begins Boot execution at real address 0 of Boot space, identified by TG<2:0>=5, following release of $\overline{RESET}$.

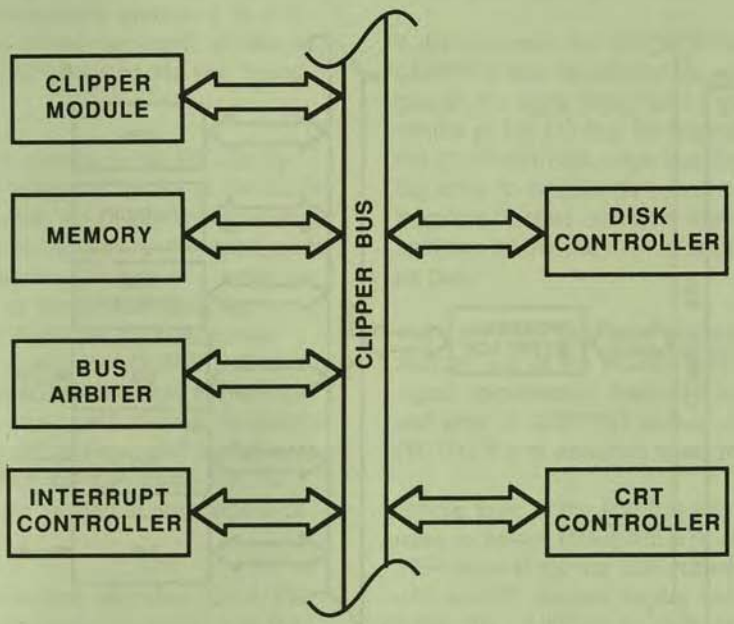
CLIPPER Modules typically boot from ROM located in Boot space, as described. However, booting from RAM located in either Boot space or Main Memory space is sometimes preferred. Boot RAM allows the flexibility of changing the Boot program without changing hardware (ROM code), but requires that the program be downloaded into the Boot RAM before system booting begins.

If the Boot RAM is located in main memory, system logic must decode TG<2:0>=5 as Main Memory space rather than Boot space. Booting then begins at location 0 of main memory space. Care must be taken, however, to ensure that the boot program does not overlap the Exception Vector Table, which must begin at location 108 hex of main memory, or with other dedicated areas of main memory. This overlap can be avoided by partitioning the Boot RAM into one area located in addresses 0 to 107 hex, and another located in available higher-address locations. The boot program can then force a jump from the lower-address Boot RAM area to the higher-address area, bypassing dedicated main memory locations.

8.3 I/O SUBSYSTEM

The CLIPPER Module has set a new standard in microprocessor-based system performance. This increased level of performance challenges the systems designer to develop interfaces between the high-speed module and the lower-speed I/O that minimize the impact of I/O function on total system performance.

Figure 8.2 Traditional I/O Implementation



The traditional approach to I/O implementation in microprocessor-based systems has been to interface the I/O directly to the system bus. Using this approach, the CLIPPER systems designer attaches I/O peripherals (e.g., graphics controllers, disk controllers, LAN controllers) on the CLIPPER Bus as shown in Figure 8.2. This may be suitable for many systems, but the resulting demands on the CLIPPER Bus may be prohibitive for some applications. One strategy for minimizing bus overhead is through the use of IOPs. The system designer should choose the appropriate I/O strategy (directly attached peripherals, IOPs, or combinations of both) for the intended application.

8.3.1 IOPs

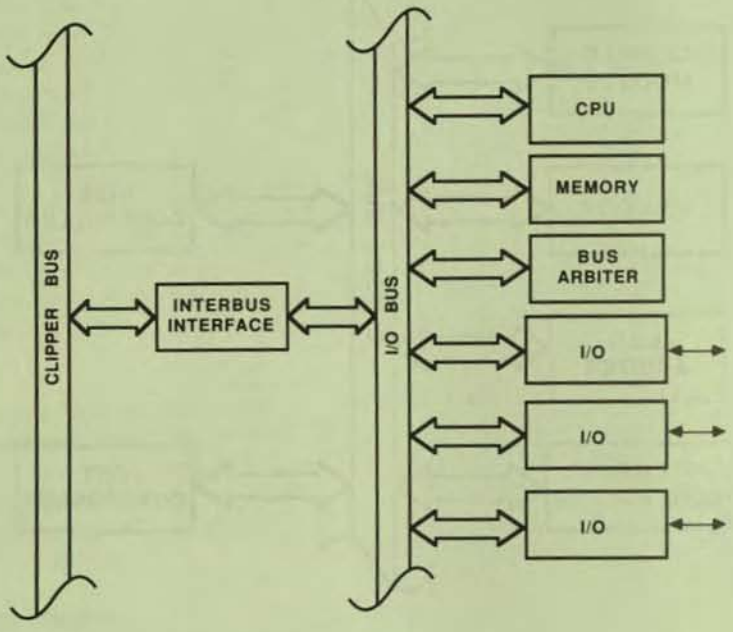
An IOP consists of a subsystem interfaced to the CLIPPER Bus that executes I/O tasks for the CLIPPER. The subsystem can be a complete, standalone system with a processor, memory, I/O peripherals, a bus arbiter, and an I/O bus. However, to the CLIPPER system, the IOP is simply an I/O device with no special bus protocol or addressing requirements. IOP architectures can take many forms and should be designed according to the intended application. An example IOP is shown in Figure 8.3.

An IOP allows true CLIPPER-I/O parallel operation. It can also allow the use of a standard bus (e.g., VME, Q-bus, MULTIBUS) for the local I/O bus.

The IOP directly controls many of the tasks associated with the CLIPPER System I/O, relieving the CLIPPER of these overhead tasks. The IOP controls access to remote peripherals such as disk drives, and manages data transfer to *IOP* buffer memory. The CLIPPER need only initialize the IOP, and support the transfer of data between the IOP and CLIPPER main memory. CLIPPER CPU and CLIPPER Bus support of I/O operations is therefore minimal, freeing the CLIPPER for its primary tasks.

The I/O bus can be a standard bus, allowing use of a family of compatible CPUs and I/O

Figure 8.3 IOP



peripherals in the IOP, with little or no interface logic. The IOP may be an existing system, already in use, significantly shortening system development time. More than one IOP can be interfaced to the CLIPPER Bus, and combinations of I/O controllers interfaced directly to the CLIPPER Bus and IOPs can be used.

8.3.2 Interbus Interface

The interbus interface, which connects the I/O bus to the CLIPPER Bus, can be either unidirectional or bidirectional, and can be buffered.

A bidirectional interface allows both the CLIPPER Module to gain mastership of the I/O bus for direct access of IOP resources, and the IOP to gain mastership of the CLIPPER Bus. A unidirectional interface allows IOP mastership of the CLIPPER Bus only.

The bidirectional interface offers a bit more flexibility than the unidirectional interface, but requires more logic. The unidirectional interface can execute all the functions of a bidirectional interface, with much less logic. This is because the unidirectional interface can be designed to accept requests from the CLIPPER for reads of CLIPPER memory by the IOP. Once the interface receives such a request, it can interrupt the IOP with a "read CLIPPER memory" interrupt request. The effect is a write by the CLIPPER to the IOP, executed as a read by the IOP.

Both bidirectional and unidirectional interbus interfaces can incorporate a buffer which buffers data from a master on one bus for later transfer to the other bus. The interface logic transfers the buffered data either independently when the buffer is full, or when the interface receives a "flush" command from a bus master.

8.3.3 Interbus Arbitration

Use of an IOP constitutes a multiple bus system consisting of the CLIPPER Bus and the I/O bus. In the multiple bus system, operations on the CLIPPER Bus by CLIPPER Bus masters, and

operations on the I/O bus by IOP bus masters are normally independent of each other: each bus serves a distinct, stand-alone system. It is only when a bus master on one bus gains access to the other bus that operations on the two buses are related.

The CLIPPER gains access to the I/O bus by first requesting and obtaining control of the CLIPPER Bus in the normal way. Once the module gains control of the bus it asserts the appropriate address, control signals, and data (if a write) on the CLIPPER Bus as usual, then waits for Ready. To the CLIPPER, the I/O bus access operation is simply a standard CLIPPER Bus operation with the requirements that the Memory Space System Tag must be 4 (TG=4) to identify the operation as an I/O access, and the address must decode to select the IOP. (Note that the CAMMU registers are located in real page 0 of I/O space).

The interbus interface logic decodes TG=4, the tag indicating noncacheable I/O space, and the IOP address as a request for the I/O bus. When the interface detects an IOP access, it asserts a bus request to the IOP bus arbiter for control of the I/O bus. Once the interbus interface receives a bus grant from the IOP bus arbiter, it asserts the CLIPPER Bus address, data (if a write), and appropriate control signals (Read/Write) on the I/O bus, then waits for Ready from the addressed I/O bus device indicating transfer of data. The interface then asserts data (if a read) and Ready on the CLIPPER Bus, allowing termination of the operation by the CLIPPER Module.

The procedure is similar for IOP accesses of CLIPPER Module resources. An I/O bus master must first request and gain access to the I/O bus, then must indicate a request to the interbus interface for access to the CLIPPER Bus, using a means suited for the particular I/O bus. The interbus interface then gains control of the CLIPPER Bus and carries out the data transfer.

8.3.4 Deadlock

A condition known as *deadlock* can occur in bidirectional multiple bus systems. Deadlock results from overlapping requests from bus masters on each bus for interbus access.

If, for instance, the CLIPPER has control of the CLIPPER Bus for attempted access of the I/O bus at the same time that an I/O bus master has control of the I/O bus for attempted access of the CLIPPER Bus, each bus master waits for the other to release its bus. Each bus master is therefore "locked out" from the other bus until one of the masters is forced to release its bus.

The CLIPPER facilitates recovery from Deadlock through use of the $\overline{\text{MSBE}}/\overline{\text{RETRY}}$ signal. This signal, explained in Section 7.2.8, forces abort and retry of CLIPPER access of I/O space (TG=4) if it is asserted while Ready is inactive.

Simple logic in the interbus interface logic can be used to assert $\overline{\text{MSBE}}/\overline{\text{RETRY}}$ whenever a CLIPPER request for the IOP occurs simultaneously with an IOP request for the CLIPPER Bus. This forces the CLIPPER to abort its bus operation and release the CLIPPER Bus, then re-arbitrate access to the CLIPPER Bus for a retry of the aborted operation. The IOP can gain access to the CLIPPER Bus after the abort by the module but before the retry, thus eliminating the Deadlock condition. The CLIPPER then waits for completion of the interbus operation by the IOP before gaining access to the I/O bus for the retry.

8.4 BUS CONTROL AND INTERRUPT LOGIC

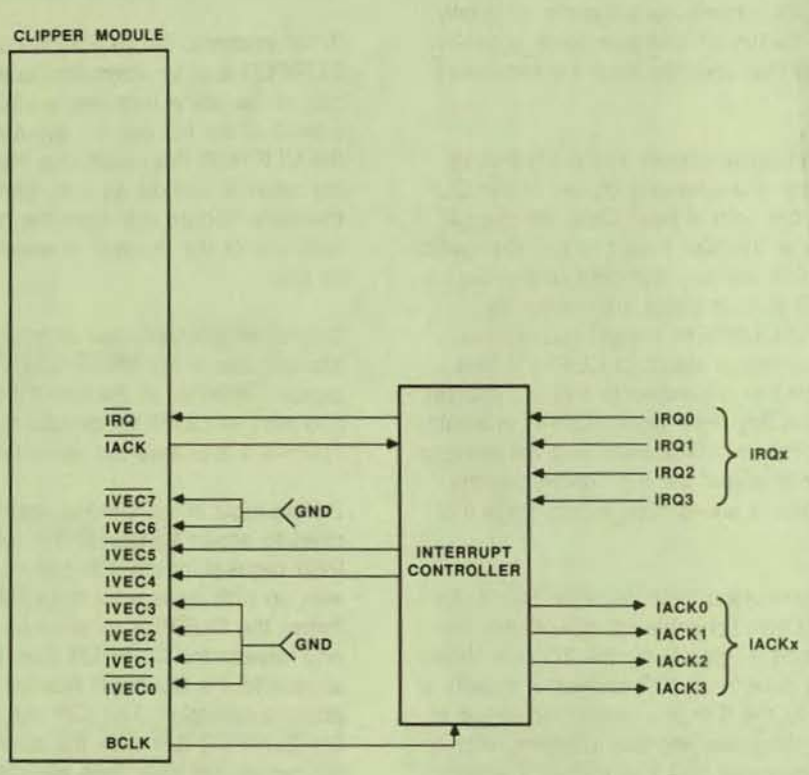
The CLIPPER Bus itself requires logic to control its capabilities. Interrupt logic, bus arbitration logic, and transceivers may be required in the CLIPPER system.

8.4.1 Interrupt Controller

As already discussed in Chapter 7, the CLIPPER CPU recognizes 255 separate maskable interrupts grouped into 16 prioritized levels. The interrupt vectors are prioritized in descending order as seen on the active low $\overline{\text{IVEC}}$ bus, such that vector 0 has the highest priority.

Since the CPU has only one $\overline{\text{IRQ}}$ line and reads only one vector number via the $\overline{\text{IVEC}}$ bus in response to assertion and release of $\overline{\text{IRQ}}$, an

Figure 8.4 Interrupt Controller



interrupt controller must be implemented in systems supporting two or more bus device IRQx interrupts. The controller arbitrates in the case of multiple interrupt requests and asserts a single vector number on the $\overline{\text{IVEC}}$ bus corresponding to the selected interrupt.

The CLIPPER Bus supports a wide variety of interrupt controller implementations. An example of a simple interrupt controller, illustrating the basic concepts of interrupt operations on the CLIPPER Bus, is shown in Figure 8.4. The example is a four channel controller with four IRQx interrupt request inputs from devices on the CLIPPER Bus, and four corresponding IACKx interrupt acknowledge outputs. The controller also has an $\overline{\text{IRQ}}$ output to the CLIPPER CPU, an $\overline{\text{IACK}}$ input from the CPU, a BCLK input from the CLIPPER Module for synchronization, and $\overline{\text{IVEC}}$ outputs.

Since the controller supports only four interrupts, only two $\overline{\text{IVEC}}$ lines are used to identify the interrupting device: $\overline{\text{IVEC5}}$ and $\overline{\text{IVEC4}}$. These lines were chosen randomly from the upper four $\overline{\text{IVEC}}$ lines to allow prioritization and selective masking of the interrupts, which is based on the states of $\overline{\text{IVEC7}}$-$\overline{\text{IVEC4}}$ and the contents of the System Status Word IL field (see Section 2.2.4). $\overline{\text{IVEC7}}$, $\overline{\text{IVEC6}}$ and $\overline{\text{IVEC3}}$-$\overline{\text{IVEC0}}$ are arbitrarily driven low to force vector numbers CF, DF, EF, and FF according to the states of $\overline{\text{IVEC5}}$ and $\overline{\text{IVEC4}}$.

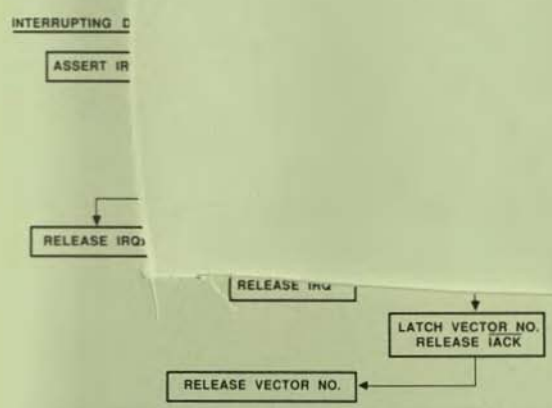
8.4.1.1 Interrupt Controller Operation

Operation of the interrupt controller begins with the assertion of one or more IRQx signals. The controller synchronizes the IRQx requests to BCLK, asserts $\overline{\text{IRQ}}$ to notify the CLIPPER Module CPU that an interrupt request is pending, then asserts the vector number ($\overline{\text{IVEC5}}$ and $\overline{\text{IVEC4}}$) which corresponds to the asserted IRQx

with the highest priority. The CPU then drives IACK to indicate that it is ready to service the interrupt, allowing the controller to assert the IACKx signal corresponding to the IRQx interrupt being service

The device w
must then rel
assertion of i
release IRQ.
vector numb
bus. The CF
releases IAC
controller is
request.

The followir



This sequence is typical, based on CLIPPER CPU requirements. The CPU, however, allows much flexibility in interrupt controller response times. Consequently controllers of various speeds can be utilized based on the number of IRQx interrupts supported and system performance requirements.

8.4.1.2 Interrupt Controller Timing Considerations

As already explained, the CLIPPER Module CAMMUs and CPU sample and assert signals on the rising edge of BCLK. For this reason, the interrupt controller must assert IRQ with sufficient setup time with respect to a rising edge of BCLK to assure recognition by the CPU on the reference edge. This implies, conversely, that the

nly
ation
free
sur-
upt
igure

erted

is no

the

he
nds
ent
riving

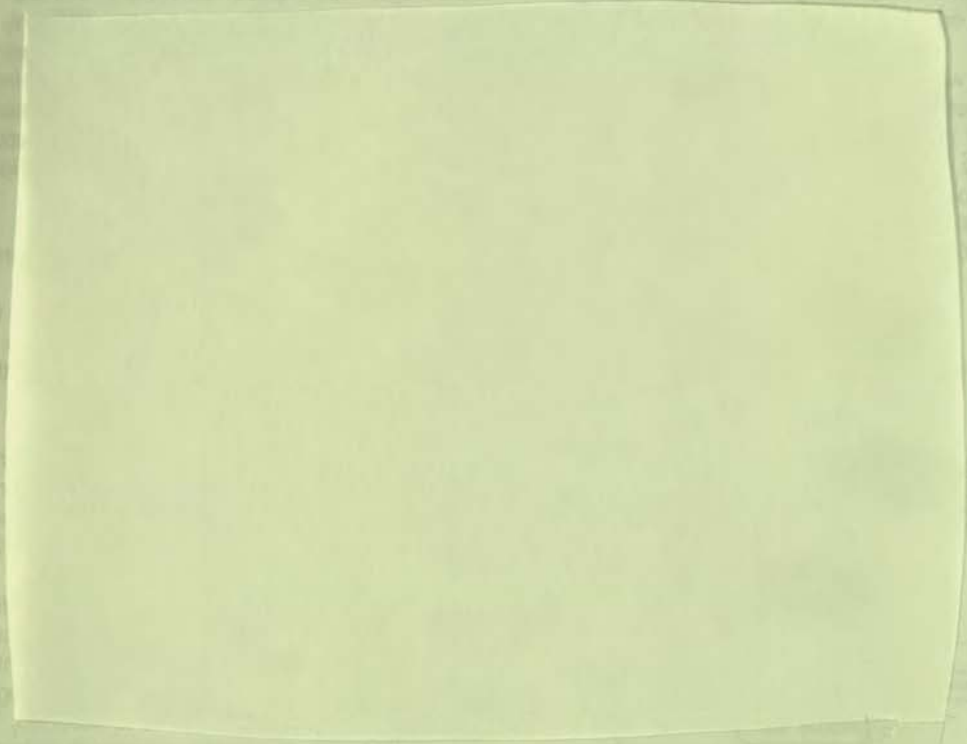
IRQx
allow
nter-
ge to

. This means that once the vector number is asserted, it can change until IRQ is released if IVEC7-IVEC4 change to a higher value. However, the changes must meet the setup time with respect to BCLK rising edges as specified in the *CLIPPER 32-Bit Microprocessor Product Description.*

Once the CPU asserts IACK, the interrupt controller responds by asserting the appropriate IACKx signal. The controller can introduce as much delay in asserting IACKx as it requires. Once the interrupting device detects its IACKx signal asserted, it releases its IRQx. The interrupting device can introduce as much delay in releasing its IRQx as it requires. The controller then releases IRQ with as much delay as it requires.

The CPU latches the interrupt vector at the rising edge of BCLK immediately following release of

Figure 8.5 Interrupt Controller Timing Example



$\overline{IRQ}$, then releases $\overline{IACK}$ as already discussed. The interrupt controller must hold $\overline{IRQ}$ in the released state for one BCLK period following release of $\overline{IACK}$ before asserting $\overline{IRQ}$ again in response to a new interrupt.

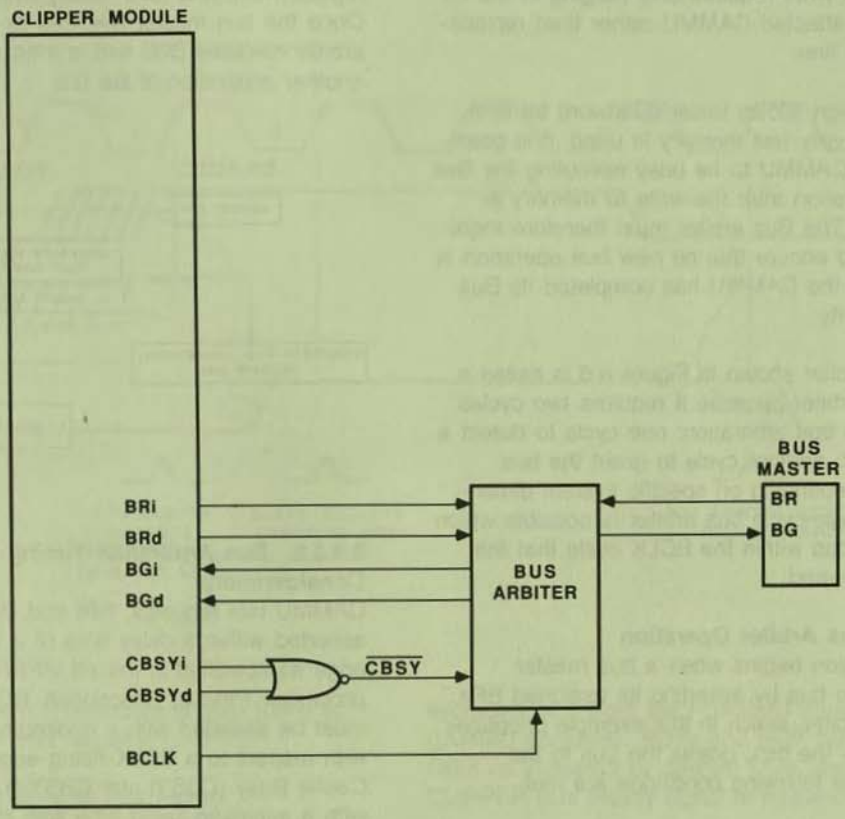
8.4.1.3 Interrupt Controller Timing Example
Timing for the four-channel interrupt controller described in this chapter is shown in Figure 8.5. In the example, IRQ3 is asserted by a bus device during the first cycle to initiate interrupt servicing. $\overline{IRQ}$ and the vector number ($\overline{IVEC5}$ and $\overline{IVEC4}$) identifying IRQ3 are asserted by the controller the following cycle.

However, IRQ0 is asserted before $\overline{IRQ}$ is released. Since this interrupt has a higher priority than IRQ3, the controller changes the vector number during the third cycle to identify IRQ0 as requiring servicing. The device asserting IRQ3 holds the interrupt request asserted until is recognized after completion of IRQ0 servicing.

The operation continues with assertion of $\overline{IACK}$ by the CPU, followed one cycle later by assertion of IACK0 by the interrupt controller. The device requesting IRQ0 servicing then releases IRQ0, forcing the interrupt controller to release $\overline{IRQ}$ one cycle later. The CPU latches the vector number upon detection of released $\overline{IRQ}$, and releases $\overline{IACK}$ one cycle later. The controller releases IACK0 one cycle after release of $\overline{IACK}$.

During this time, IRQ3 servicing is pending. The

Figure 8.6 Bus Arbiter



controller keeps $\overline{IRQ}$ release for 1 cycle after release of $\overline{IACK}$ as required, then begins servicing of IRQ3 as shown.

8.4.2 CLIPPER BUS ARBITER

The CLIPPER system bus arbiter must be capable of receiving bus requests (BRx) from the I-CAMMU, the D-CAMMU, and other bus masters. It may support priority bus assignment such that in cases of multiple requests for the bus, it grants the bus to the highest-priority requesting device. It must assert a unique bus grant (BGx) for each bus master which is granted the bus.

Implementation of a two-cycle, three-master bus arbiter is shown in Figure 8.6. The arbiter has inputs from each of the CAMMUs (BRi and BRd), and from one other bus master (BR). The arbiter also has a Cache Busy ($\overline{CBSY}$) input used to inhibit granting of the bus to a new master while a CAMMU is executing Bus Watch, and a BCLK input for synchronization. The arbiter outputs are bus grant signals to each of the three bus masters: BGi, BGd, and BG.

$\overline{CBSY}$ is monitored by the bus arbiter for use in systems utilizing very fast memory which allow completion of quadword writes without regard to CAMMU Bus Watch activity. If the memory interface always waits until $\overline{CBSY}$ is inactive (low) before asserting RDY, then the bus arbiter need not monitor $\overline{CBSY}$.

The memory interface can be designed to latch

written quadword data and assert $\overline{RDY}$ without regard to the state of $\overline{CBSY}$, since a cache hit by a quadword write requires only purging of the hit line by the affected CAMMU rather than replacement of the line.

Such a design allows faster quadword transfer. However if very fast memory is used, it is possible for the CAMMU to be busy executing the Bus Watch operation after the write to memory is completed. The Bus arbiter must therefore monitor $\overline{CBSY}$ to ensure that no new bus operation is begun until the CAMMU has completed its Bus Watch activity.

The bus arbiter shown in Figure 8.6 is called a two-cycle arbiter because it requires two cycles to complete bus arbitration: one cycle to detect a bus request; another cycle to grant the bus. However, depending on specific system dimensions, a single-cycle bus arbiter is possible which grants the bus within the BCLK cycle that the bus is requested.

8.4.2.1 Bus Arbiter Operation

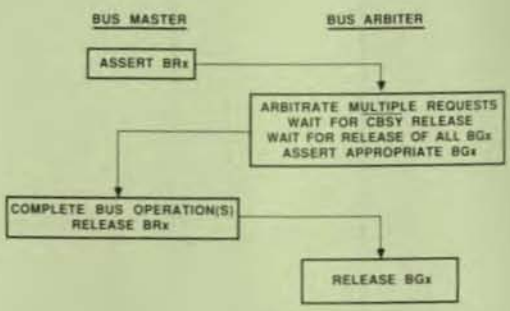
Bus arbitration begins when a bus master requests the bus by asserting its assigned BRx. The bus arbiter, which in this example prioritizes requests for the bus, grants the bus to the master if the following conditions are met:

1. No higher priority BRx is asserted.
2. All Bus arbiter BGx are released.
3. The Cache Busy line is not active.

These conditions assure that bus requests are honored on a priority basis, that a bus master can complete its bus operation(s) once it has been granted the bus, and that the CAMMUs are allowed to complete Bus Watch activity before the bus is committed to new operations.

The bus arbiter then grants the bus to the priority requesting bus master by asserting the appropriate BGx.

Once a bus master has asserted its bus request, it must hold its BRx asserted until it has completed all of its data transfers. This ensures that the master retains continuous control of the bus throughout the multiple bus operations required in cache line replacement, DTU page replacement, and read-modify-write operations. Once the bus master releases its BRx, the bus arbiter releases BGx and is then ready for another arbitration of the bus.



8.4.2.2 Bus Arbitration Timing Considerations

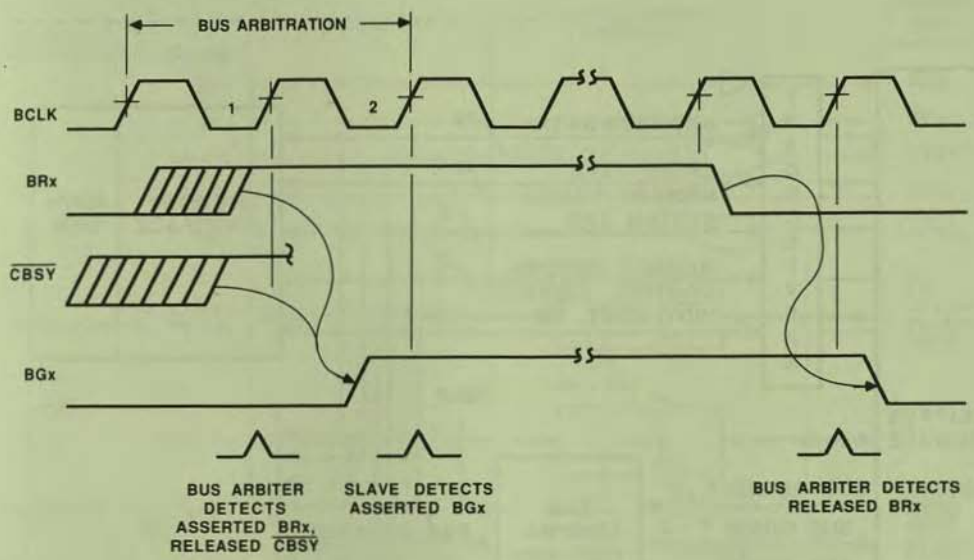
CAMMU bus requests, BRi and BRd, are asserted within a delay time of a BLCK rising edge as specified in the *CLIPPER 32-Bit Microprocessor Product Description*. BGi and BGd must be asserted with a minimum setup time with respect to a BCLK rising edge. CAMMU Cache Busy (CBSYi and CBSYd) are asserted with a minimum setup time with respect to a BCLK rising edge, and released within a specified delay time with respect to a BCLK rising edge. These timing constraints are based on CLIPPER Bus specifications.

8.4.2.3 Bus Arbitration Timing Example

Figure 8.7 shows timing for the bus arbiter depicted in Figure 8.6. When the arbiter detects one or more Bus Requests active and no $\overline{CBSY}$ or Bus Grant signals active during a cycle, the arbiter asserts a Bus Grant to the highest-priority requesting device following the next rising edge of BCLK. Thus a Bus Request cycle can be followed immediately by a Bus Grant cycle, resulting in two-cycle arbitration.

The bus master releases the bus following completion of its bus operation by releasing BRx, forcing the bus arbiter to release BGx the follow-

Figure 8.7 Bus Arbiter Timing Example



ing cycle as shown. The arbiter is then ready to arbitrate the next bus request.

8.4.3 CLIPPER Module Interfacing

CLIPPER Module output signals can be interfaced through tranceivers as shown in Figure 8.8, if necessary, to mininize loading of the module output drivers and thereby allow high-speed module operation. Some signals may be interfaced through combinational logic according to system requirements.

Typical CLIPPER Module interfacing is shown in Figure 8.9. Note that CBSYi and CBSYd are tied to a NOR gate whose output is tied to the CLIPPER Bus $\overline{CBSY}$ line. System logic (memory interface and bus arbiter) functions identically for asserted CBSYi and CBSYd, so only one NOR'd $\overline{CBSY}$ signal is required on the bus. Note also that system $\overline{RDY}$ and I-CAMMU $\overline{RDYoi}$ are NANDed to form a common RDYi input to the D-CAMMU. This allows the D-CAMMU to respond to Ready from bus slaves during bus slave accesses, and from the I-CAMMU during I-CAMMU accesses. $\overline{RDYo}$ and $\overline{RDY}$ can be wire-OR'd as indicated by dashed lines since only one CLIPPER Bus Ready signal is required.

The multiplexed Address/Data signals AD<31:0> are interfaced through transceivers, with directional control for the transceivers provided by the open drain CAMMU $\overline{DIR}$ signals. The drivers assert address onto the bus during each bus operation; then drive data onto the bus if a write operation, or drive data to the CAMMUs if a read.

TG<2:0>, CT<5:0>, $\overline{TR}$ and $\overline{LOCK}$ are also interfaced through transceivers, with directional control for these transceivers derived from the CAMMU BR and BG signals. Normally the drivers drive to the CAMMUs, allowing the CAMMUs to monitor the signals for Bus Watch operation. When either CAMMU receives a bus grant in response to a bus request, the transceivers drive the signals toward the bus.

Figure 8.8 CLIPPER System – Block Level

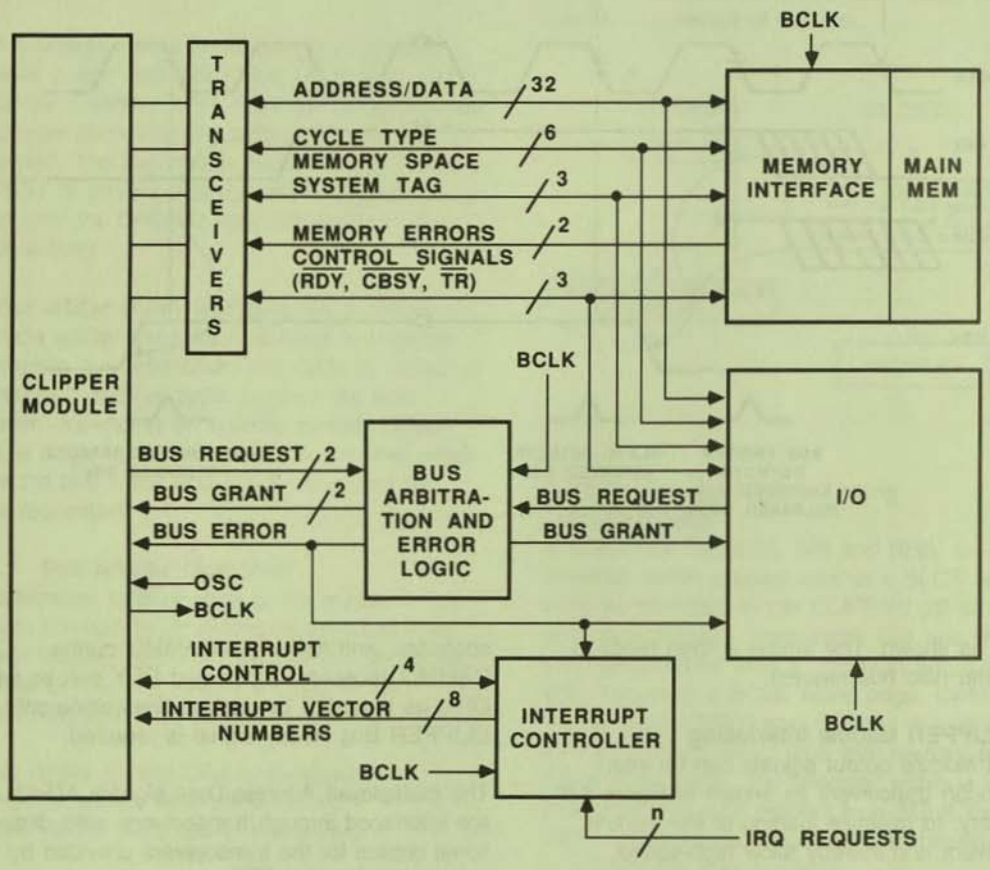
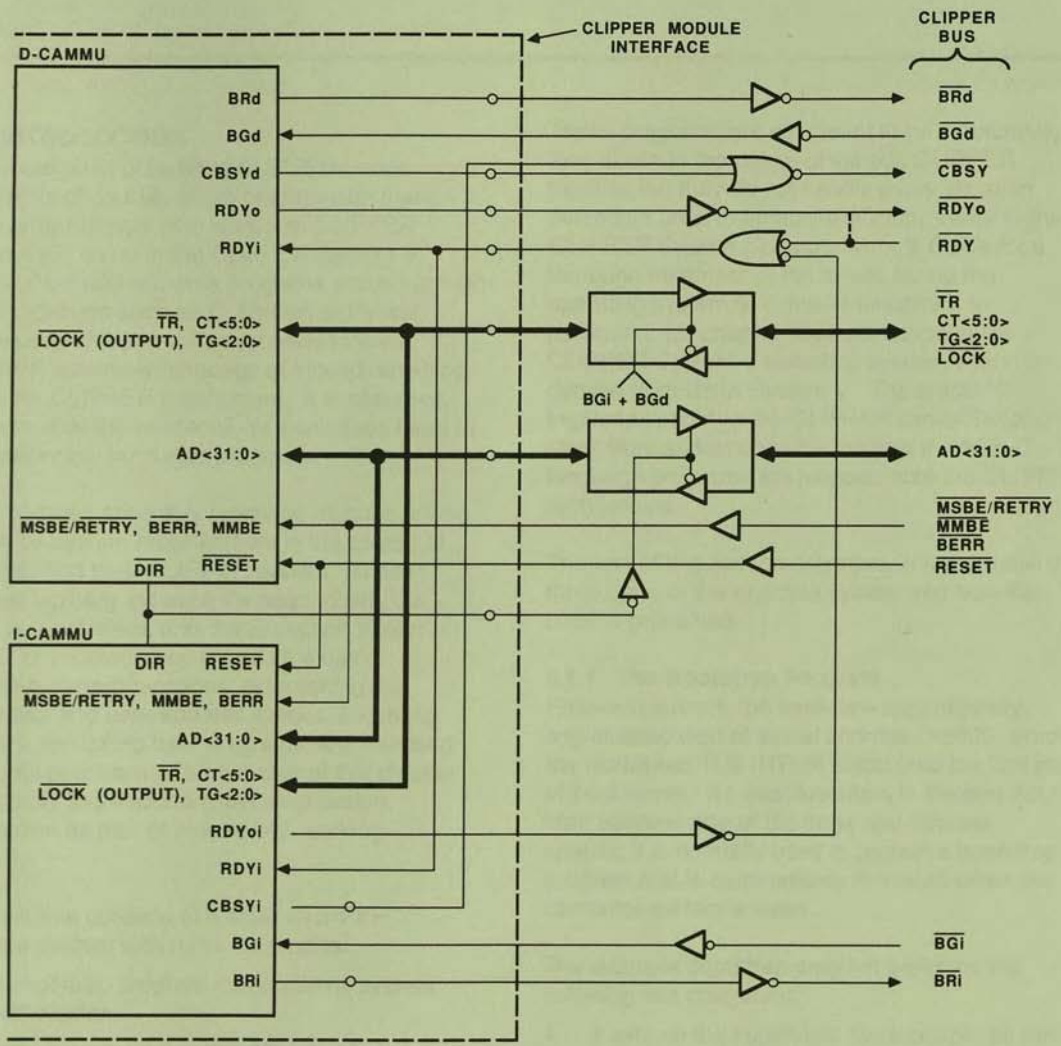


Figure 8.9 CLIPPER Module Buffering



NOTES:

1. Only signals between CAMMU and CLIPPER bus are shown. This does not include interrupt signals. BCLK, URF, URDIAG, or OSC.
2. RDY and RDYo can connected together on the CLIPPER bus.
3. —o— pullup resistor.

Chapter 9
Software Design

9.1 INTRODUCTION

The whole point of building CLIPPER-based systems is of course, to run programs on them. Many programmers who work with CLIPPER systems will do so in the CLIPPER System V environment and will write programs entirely in high-level languages such as C, Fortran or Pascal. These programmers will never need to know CLIPPER assembly language or indeed, anything about the CLIPPER architecture. It is assumed, however, that the reader of this book does need to write assembly language programs.

Many of these assembly language routines will be written by system programmers in the course of implementing their CLIPPER system. These routines typically initialize the page tables, the processor registers, and the Exception Vector Table. In addition, they provide the basic operating system functions: establishing the supervisor and user address spaces, allocating memory, executing user programs, and handling interrupts and traps. The purpose of this chapter is to supply this important low-level design information as part of a complete, working example.

The example consists of a small CLIPPER software system with three main parts:

1. A bootstrap program that performs system initialization.
2. A simple supervisor (the Monitor) that establishes a virtual memory environment, maintains the system clock, processes exceptions, executes user programs, and handles several system calls that can be made from the user programs.
3. A user program that compares two different algorithms for generating a bounded sequence of Fibonacci numbers. This program makes several system calls to the Monitor and also illustrates the use of shared memory.

These programs are not meant to be exhaustive; they illustrate the usage of various CLIPPER facilities but they do not handle every situation that might arise in an actual system. Refer to the *CLIPPER System Software Writer's Guide* for a thorough treatment of the issues facing the operating system or compiler designer. In particular, this chapter does not describe the CLIPPER System V operating system, which is derived from UNIX System V. The article "C Implementation" in the *CLIPPER Cross Support User Manual* describes the manner in which C language programs are mapped onto the CLIPPER architecture.

The rest of this section describes in more detail the three parts of the example system and how the code is presented.

9.1.1 The Bootstrap Program

Following a reset, the hardware automatically begins execution at virtual address 0x6000, which the Hardwired TLB (HTLB) maps onto the first page of boot space. As was described in Section 3.2.1, boot space is one of the three real address spaces; it is normally used to contain a bootstrap program that is automatically activated when the computer system is reset.

The example bootstrap program performs the following five operations:

1. It sets up the supervisor stack pointer so that boot space page 1 is used as a temporary supervisor stack (the monitor will establish a permanent supervisor stack later).
2. It clears the Program Status Word (PSW) and the System Status Word (SSW). This has the effect of leaving the CLIPPER in supervisor mode with memory mapping disabled.
3. It initializes the Instruction and Data CAMMUs so that unmapped mode addresses are noncacheable and bus watch is enabled. In

addition, it enables prefetch for the Instruction CAMMU only.

4. It sets all of the vectors in the Vector Table to point to a null interrupt handler.
5. It loads a supervisor mode program (the Monitor) and transfers control to it.

9.1.2 The Monitor

The example Monitor is basically a primitive operating system. It runs in supervisor mode and has control of all system resources. The Monitor performs a number of functions, in particular:

- setting up the virtual-to-real address map for the Monitor and enabling mapping
- setting up the Vector Table to point to routines for handling system calls, traps, and interrupts
- moving the supervisor stack to high memory
- setting up and managing an interrupt-driven system clock
- providing a command interpreter for a few simple commands entered through a character input device
- setting up the user program's address map
- providing system call services to the user program for allocating available memory and for transferring characters to and from the character-oriented I/O devices.

9.1.3 The User Program

The example user program calculates the first *N* Fibonacci numbers using two different algorithms, then prints the amount of time each algorithm required. (The Fibonacci numbers are defined by a sequence, 0, 1, 1, 2, 3, 5, 8 ..., where each element in the sequence is the sum of the previous two elements.)

The program uses system calls to acquire pages of memory as needed and to display the elapsed time for each algorithm. It retrieves the value of the system clock from a global data area in shared memory.

9.1.4 How the Code is Presented

The example code is broken into sections, each of which fits on one page of the manual. The code sections are shown on odd-numbered pages; the accompanying even-numbered pages contain narration that describes the design principles, algorithms, and implementation techniques used in the particular code sections. The narration also describes the limitations of the code. A cross reference listing of both the Monitor and the user program appears in Section 9.6, following the examples.

The code in this chapter makes use of the following assembler directives:

.align *integer_expression*
Aligns the following text or data so that it begins on an *integer_expression*-byte boundary, where *integer_expression* evaluates to 1, 2, 4, or 8.

.byte *integer_expression* {, *integer_expression*}
Allocates an 8-bit byte for each *integer_expression*, and initializes it to contain *integer_expression* as its value.

.data
Places all of the following code up to the next **.text** directive into the program's data segment.

.double *float_expression* {, *float_expression*}
Allocates a 64-bit double word for each *float_expression*, and initializes it to contain *float_expression* as its value.

.globl *ident*
Declares *ident* as a global symbol. If the current module defines *ident*, then it becomes a global definition. Otherwise, it becomes an external request.

.long *integer_expression* {, *integer_expression*}
Allocates a 32-bit word for each *integer_expression*, and initializes it to contain *integer_expression* as its value.

.org *integer_expression*
Moves the location counter forward to *integer_expression*. *Integer_expression* must evaluate to a relocatable address in the current segment. Subsequent code or data is stored beginning at that address. *Integer_expression* may not be absolute, nor may it cause the location counter to move backwards.

.set *ident, integer_expression*
Defines or redefines *ident* with the value *integer_expression*. *Ident* must be non-null.

.space *integer_expression*
Allocates *integer_expression* bytes of storage. *Integer_expression* must be greater than zero. The allocated storage is not initialized.

.text
Places all of the following code up to the next **.data** directive into the program's text segment.

.word *integer_expression* {, *integer_expression*}
Allocates a 16-bit "word" (CLIPPER halfword) for each *integer_expression*, and initializes it to contain *integer_expression* as its value.

9.1.5 Clipper Register Definitions

This section contains the assembly language definitions for every field in the various CLIPPER registers. Also included here are the definitions for the fields within the CLIPPER Virtual Address. These definitions are used in the example programs throughout this chapter.

9.1.5.1 Program Status Word (PSW)

The listing on the following page contains definitions for the fields in the Program Status Word (**PSW**).

The definitions on lines 26 through 28 make it easy to manipulate the floating-point trap enable and status bits. The code fragment in example 9.1 tests for the presence of any floating-point exceptions, while Example 9.2 enables all of the floating-point traps.

The shift defintions on lines 43 and 44 are used to extract the CPU or Memory Trap Status value. The code in Example 9.3 converts the CPU Trap Status field into an integer.

Example 9.1

```
movpw   psw,r0              # R0 = current PSW
andi    $PSW_FTS,r0         # Are there any floating-point exceptions?
brne    FPError             #  Yes - go process them
```

Example 9.2

```
movpw   psw,r0              # R0 = current PSW
ori     $PSW_EFT,r0         # Enable all floating-point traps
movwp   r0,psw              # Rewrite the PSW
```

Example 9.3

```
movpw   psw,r0              # R0 = current PSW
andi    $PSW_CTS,r0         # R0 = CPU Trap Status bits
shli    $-SHF_PSW_CTS,r0    # R0 = Integer CPU Trap Status
```

```
0001: #
0002: # Program Status Word.
0003: #
0004:     .set   PSW_N,           0x00000001    # Negative
0005:     .set   PSW_Z,           0x00000002    # Zero
0006:     .set   PSW_V,           0x00000004    # Overflow
0007:     .set   PSW_C,           0x00000008    # Carry
0008:     .set   PSW_FX,          0x00000010    # Floating Inexact
0009:     .set   PSW_FU,          0x00000020    # Floating Underflow
0010:     .set   PSW_FD,          0x00000040    # Floating Divide by Zero
0011:     .set   PSW_FV,          0x00000080    # Floating Overflow
0012:     .set   PSW_FI,          0x00000100    # Floating Invalid Operation
0013:     .set   PSW_EFX,         0x00000200    # Enable Floating Inexact Trap
0014:     .set   PSW_EFU,         0x00000400    # Enable Floating Underflow Trap
0015:     .set   PSW_EFD,         0x00000800    # Enable Floating Divide by 0 Trap
0016:     .set   PSW_EFV,         0x00001000    # Enable Floating Overflow Trap
0017:     .set   PSW_EFI,         0x00002000    # Enable Floating Invalid Op Trap
0018:     .set   PSW_EFT,         0x00004000    # Enable Floating Trap
0019:     .set   PSW_FR,          0x00018000    # Floating Rounding Mode
0020:     .set   PSW_RSV1,        0x007E0000    # Reserved Bits
0021:     .set   PSW_T,           0x00800000    # Trace Trap
0022:     .set   PSW_CTS,         0x0F000000    # CPU Trap Status
0023:     .set   PSW_MTS,         0xF0000000    # Memory Trap Status
0024:
0025: # Floating Trap Status Bits.
0026:     .set   PSW_FTS,         0x000001F0    # Floating Trap Status Bits
0027:     .set   PSW_FPB,         0x0001FFF0    # All Floating-Point Bits
0028:     .set   PSW_FPE,         0x00007E00    # All Floating-Pt Trap Enables
0029:
0030: # CPU Trap Status Bits.
0031:     .set   PSW_CTS_IDIV0,   0x02000000    # Integer Divide by 0
0032:     .set   PSW_CTS_ILLOP,   0x04000000    # Illegal Operation
0033:     .set   PSW_CTS_PRVIN,   0x05000000    # Privileged Instruction
0034:     .set   PSW_CTS_TRACE,   0x07000000    # Trace Trap
0035:
0036: # Memory Trap Status Bits.
0037:     .set   PSW_MTS_CME,     0x10000000    # Corrected Memory Error
0038:     .set   PSW_MTS_UCME,    0x20000000    # Uncorrectable Memory Error
0039:     .set   PSW_MTS_PGFLT,   0x50000000    # Page Fault
0040:     .set   PSW_MTS_RXFLT,   0x60000000    # Read/Execute Protection Flt
0041:     .set   PSW_MTS_WRFLT,   0x70000000    # Write Protection Fault
0042:
0043:     .set   SHF_PSW_CTS,     24            # Shift value for PSW_CTS
0044:     .set   SHF_PSW_MTS,     28            # Shift value for PSW_MTS
```

9.1.5.2 System Status Word (SSW)

The listing on the following page contains definitions for the fields in the System Status Word (**SSW**). The **SSW_RSV1** field contains bits that are reserved to Fairchild for future expansion. You should set them to zero when writing the **SSW**.

```
0045: #
0046: # System Status Word.
0047: #
0048:     .set   SSW_IN,          0x0000000F    # Interrupt Number
0049:     .set   SSW_IL,          0x000000F0    # Interrupt Level
0050:     .set   SSW_EI,          0x00000100    # Enable Interrupts
0051:     .set   SSW_RSV1,        0x003FFE00    # Reserved Bits
0052:     .set   SSW_FRD,         0x00400000    # Floating Registers Dirty
0053:     .set   SSW_TP,          0x00800000    # Trace Trap Pending
0054:     .set   SSW_ECM,         0x01000000    # Enable Corrected Memory Err
0055:     .set   SSW_RSV2,        0x02000000    # Reserved Bit
0056:     .set   SSW_M,           0x04000000    # Mapped Mode
0057:     .set   SSW_KU,          0x08000000    # User Protect Key
0058:     .set   SSW_UU,          0x10000000    # User Data Mode
0059:     .set   SSW_K,           0x20000000    # Protect Key
0060:     .set   SSW_U,           0x40000000    # User Mode
0061:     .set   SSW_P,           0x80000000    # Previous Mode
0062:
```

9.1.5.3 Cache/Memory Management Unit (CAMMU)

The listing on the following page contains definitions for the registers in the Cache/Memory Management Unit (**CAMMU**).

Lines 66 through 68 define the base address of the Data, Instruction, and Global CAMMUs. Lines 71 through 76 define the offsets from those base addresses of the TLB and each of the five CAMMU registers.

The remaining lines combine the base and register offsets for each of the CLIPPER CAMMUs. The code in Example 9.4 illustrates the use of these definitions by setting up the User Page Directory Origin register in both the Instruction and Data CAMMUs.

Example 9.4

```
loada   UPageTblDir,r0      # R0 = User Page Table Directory Address
storw   r0,C_I_UPDO         # Set the user instruction Page Directory Origin
storw   r0,C_D_UPDO         # Set the user data Page Directory Origin
```

```
0063: #
0064: # CAMMU Base Addresses.
0065: #
0066:     .set   D_,              0x00004800    # Data CAMMU
0067:     .set   I_,              0x00004A00    # Instruction CAMMU
0068:     .set   G_,              0x00004C00    # Global CAMMU
0069:
0070: # CAMMU Register Offsets.
0071:     .set   _TLB,            0x00000000    # TLB Base
0072:     .set   _SPDO,           0x00000104    # Super Pg Tbl Dir Offset
0073:     .set   _UPDO,           0x00000108    # User Pg Tbl Dir Offset
0074:     .set   _VFA,            0x00000110    # Virtual Fault Address Register
0075:     .set   _CTL,            0x00000140    # Control Register
0076:     .set   _RS,             0x00000180    # Reset Register
0077:
0078: # I-CAMMU Register Addresses.
0079:     .set   C_I_TLB,         (I_+_TLB)     # I-CAMMU TLB
0080:     .set   C_I_SPDO,        (I_+_SPDO)    # I-CAMMU SPDO
0081:     .set   C_I_UPDO,        (I_+_UPDO)    # I-CAMMU UPDO
0082:     .set   C_I_VFA,         (I_+_VFA)     # I-CAMMU VFA
0083:     .set   C_I_CTL,         (I_+_CTL)     # I-CAMMU CTL
0084:     .set   C_I_RS,          (I_+_RS)      # I-CAMMU RS
0085:
0086: # D-CAMMU Register Addresses.
0087:     .set   C_D_TLB,         (D_+_TLB)     # D-CAMMU TLB
0088:     .set   C_D_SPDO,        (D_+_SPDO)    # D-CAMMU SPDO
0089:     .set   C_D_UPDO,        (D_+_UPDO)    # D-CAMMU UPDO
0090:     .set   C_D_VFA,         (D_+_VFA)     # D-CAMMU VFA
0091:     .set   C_D_CTL,         (D_+_CTL)     # D-CAMMU CTL
0092:     .set   C_D_RS,          (D_+_RS)      # D-CAMMU RS
0093:
0094: # G-CAMMU Register Addresses.
0095:     .set   C_G_TLB,         (G_+_TLB)     # G-CAMMU TLB
0096:     .set   C_G_SPDO,        (G_+_SPDO)    # G-CAMMU SPDO
0097:     .set   C_G_UPDO,        (G_+_UPDO)    # G-CAMMU UPDO
0098:     .set   C_G_VFA,         (G_+_VFA)     # G-CAMMU VFA
0099:     .set   C_G_CTL,         (G_+_CTL)     # G-CAMMU CTL
0100:     .set   C_G_RS,          (G_+_RS)      # G-CAMMU RS
0101:
```

The listing on the opposite page contains definitions for the fields in the registers of the Cache/Memory Management Unit (**CAMMU**).

Lines 103 through 107 define the fields in the CAMMU Raw TLB registers. The various bit combinations support access to the Real or Virtual address configurations of either the W or X compartment of each of the 64 possible TLB lines. The code fragment in Example 9.5 loads the Real Address information from the W compartment of TLB line set #2.

Example 9.6 loads the Virtual Address information from the W compartment of TLB line set #0x10.

Lines 111 through 116 define the fields in the CAMMU Control register. The bit combinations within the Unmapped System Tag field (**C_CTL_UST**) are further defined in lines 119 through 122. Note that these bits represent a subset of the System Tag values, defined in the Page Table Entry in Section 9.1.4.

Lines 125 through 132 define the bits within the CAMMU Reset register. **C_RS_ALL** contains all bits with in the register, so you can easily reset the entire CAMMU. The code fragment in Example 9.7 resets all User Valid lines within the CAMMUs. This code would be executed immediately before performing a context switch from the supervisor to a new user program.

Example 9.8 initializes both the instruction and data CAMMU. In both cases, the Unmapped System Tag is set to Private, Copy-back (**UST_PVCB**) and bus is enabled for CPU writes (**C_CTL_EWCW**) and I/O reads and writes (**C_CTL_EWIR** and **C_CTL_EWIW**). In addition, prefetch is enabled (**C_CTL_EP**) for the instruction CAMMU only. Note that **C_CTL_DP** is a pseudo-register value which documents the fact that prefetch is disabled for the Data CAMMU.

Example 9.5

```
loadw   (C_TLB_RA+C_TLB_XLINE+(0x02<SHF_TLB_LNSET)),r0
```

Example 9.6

```
loadw   (C_TLB_VA+C_TLB_WLINE+(0x10<SHF_TLB_LNSET)),r0
```

Example 9.7

```
loadi   $C_RS_RUV,r0        # R0 = Reset all User Valid bits
storw   r0,C_I_RS           # Reset the Instruction CAMMU
storw   r0,C_D_RS           # Reset the Data CAMMU
```

Example 9.8

```
loadi   $(C_CTL_EP|C_CTL_EWCW|C_CTL_EWIR|C_CTL_EWIW|C_CTL_UST_PVCB),r0
storw   r0,C_I_CTL          # Initialize the Instruction CAMMU
loadi   $(C_CTL_DP|C_CTL_EWCW|C_CTL_EWIR|C_CTL_EWIW|C_CTL_UST_PVCB),r0
storw   r0,C_D_CTL          # Initialize the Data CAMMU
```

```
0102: # CAMMU TLB Bits.
0103:     .set   C_TLB_RA,         0x00000000    # Real Addr select
0104:     .set   C_TLB_VA,         0x00000001    # Virt Addr select
0105:     .set   C_TLB_WLINE,      0x00000000    # W-Line select
0106:     .set   C_TLB_XLINE,      0x00000002    # X-Line select
0107:     .set   C_TLB_LNSET,      0x000000FC    # Line set select
0108:     .set   SHF_TLB_LNSET,    2             # Line set shift value
0109:
0110: # CAMMU Control Register Bits.
0111:     .set   C_CTL_EP,         0x00000001    # Enable Prefetch
0112:     .set   C_CTL_DP,         0x00000000    # Disable Prefetch (PseudoReg)
0113:     .set   C_CTL_EWCW,       0x00000002    # Enable Watch - CPU Writes
0114:     .set   C_CTL_EWIW,       0x00000004    # Enable Watch - I/O Writes
0115:     .set   C_CTL_EWIR,       0x00000008    # Enable Watch - I/O Reads
0116:     .set   C_CTL_UST,        0x00000030    # Unmapped System Tag
0117:
0118: # CAMMU Control Register Unmapped System Status Bits.
0119:     .set   C_CTL_UST_PVWT,   0x00000000    # 0: Private, Write-Thru Mem
0120:     .set   C_CTL_UST_SHWT,   0x00000010    # 1: Shared, Write-Thru Memory
0121:     .set   C_CTL_UST_PVCB,   0x00000020    # 2: Private, Copy-Back Memory
0122:     .set   C_CTL_UST_NCMM,   0x00000030    # 3: Noncacheable Main Memory
0123:
0124: # CAMMU Reset Register Bits.
0125:     .set   C_RS_RLVW,        0x00000001    # Reset "W" LV Bits in Cache
0126:     .set   C_RS_RLVX,        0x00000002    # Reset "X" LV Bits in Cache
0127:     .set   C_RS_RSV,         0x00000004    # Reset All SV Bits in TLB
0128:     .set   C_RS_RUV,         0x00000008    # Reset All UV Bits in TLB
0129:     .set   C_RS_RD,          0x00000010    # Reset All D Bits in TLB
0130:     .set   C_RS_RR,          0x00000020    # Reset All R Bits in TLB
0131:     .set   C_RS_RU,          0x00000040    # Reset All U Bits in Cache
0132:     .set   C_RS_CD,          0x00000080    # Clear Dirty
0133:
0134:     .set   C_RS_ALL,         0x000000FF    # Reset Everything
```

9.1.5.4 Page Table Entry (PTE)

The listing on the opposite page contains definitions for the fields in the Page Table Entry (**PTE**).

Lines 147 through 150 contain generic encodings for the System-Defined page type field. The example code in this chapter does not use these bits, so they are not defined further.

Lines 153 through 168 contain generic encodings for the CLIPPER protection levels. The Monitor program in section 9.4 includes mnemonic re-definitions of the protection levels.

Line 180 contains the shift value for use in extracting the Real Address field (**PTE_RA**) from the Page Table Entry. The code fragment in Example 9.9 extracts the Real Address field from the Page Table Entry for Supervisor page # 0x37.

Example 9.9

```
loadw    (0x37*4)+SPTbl,r0  # R0 = Page Table Entry for Supv page 0x37
andi     $PTE_RA,r0         # R0 = Real Address bits
shli     $-SHF_PTE_RA,r0    # R0 = Real Address
```

```
0135: #
0136: # Page Table Entries.
0137: #
0138:     .set   PTE_F,          0x00000001   # Page Fault
0139:     .set   PTE_R,          0x00000002   # Page Referenced
0140:     .set   PTE_D,          0x00000004   # Page Dirty
0141:     .set   PTE_PL,         0x00000078   # Protection Level
0142:     .set   PTE_S,          0x00000180   # System Reserved
0143:     .set   PTE_ST,         0x00000E00   # System Tag
0144:     .set   PTE_RA,         0xFFFFF000   # Real Address
0145:
0146: # System-Defined Page Type Encodings.
0147:     .set   PTE_S_00,       0x00000000   # 0: System-defined
0148:     .set   PTE_S_01,       0x00000080   # 1: System-defined
0149:     .set   PTE_S_02,       0x00000100   # 2: System-defined
0150:     .set   PTE_S_03,       0x00000180   # 3: System-defined
0151:
0152: # Protection Levels  (U,  UU,  K,  KU  States)  0X1X  0X0X  1X1X  1X0X
0153:     .set   PTE_PL_00,      0x00000000   # 0: RW     --    --    --
0154:     .set   PTE_PL_01,      0x00000008   # 1: RW     RW    --    --
0155:     .set   PTE_PL_02,      0x00000010   # 2: RW     RW    RW    --
0156:     .set   PTE_PL_03,      0x00000018   # 3: RW     RW    RW    RW
0157:     .set   PTE_PL_04,      0x00000020   # 4: RW     RW    RW    R
0158:     .set   PTE_PL_05,      0x00000028   # 5: RW     RW    R     R
0159:     .set   PTE_PL_06,      0x00000030   # 6: RW     R     R     R
0160:     .set   PTE_PL_07,      0x00000038   # 7: RWE    RWE   RWE   RWE
0161:     .set   PTE_PL_08,      0x00000040   # 8: RE     --    --    --
0162:     .set   PTE_PL_09,      0x00000048   # 9: R      RE    --    --
0163:     .set   PTE_PL_0A,      0x00000050   # A: R      R     RE    --
0164:     .set   PTE_PL_0B,      0x00000058   # B: R      R     RE    RE
0165:     .set   PTE_PL_0C,      0x00000060   # C: --     RE    --    RE
0166:     .set   PTE_PL_0D,      0x00000068   # D: --     --    RE    --
0167:     .set   PTE_PL_0E,      0x00000070   # E: --     --    --    RE
0168:     .set   PTE_PL_0F,      0x00000078   # F: --     --    --    --
0169:
0170: # System Tag Encodings.
0171:     .set   PTE_ST_PVWT,    0x00000000   # 0: Private, Write-through
0172:     .set   PTE_ST_SHWT,    0x00000200   # 1: Shared, Write-through
0173:     .set   PTE_ST_PVCB,    0x00000400   # 2: Private, Copy-Back
0174:     .set   PTE_ST_NCMM,    0x00000600   # 3: Noncacheable Main Memory
0175:     .set   PTE_ST_NCIO,    0x00000800   # 4: Noncacheable IO Space
0176:     .set   PTE_ST_NCBS,    0x00000A00   # 5: Noncacheable Boot Space
0177:     .set   PTE_ST_CPRG,    0x00000C00   # 6: Cache purge
0178:     .set   PTE_ST_NCSM,    0x00000E00   # 7: Noncacheable Slave Mode
0179:
0180:     .set   SHF_PTE_RA,     12           # Shift value for PTE_RA
0181:
```

9.1.5.5 Page Table Directory Entry (PTDE)

The listing on the opposite page contains definitions for the fields in the Page Table Directory Entry (**PTDE**).

The bits defined by the field **PTDE_RSV1** are reserved to Fairchild for future use. You should set them to zero when you write the Page Table Directory Entries.

The code fragment in Example 9.10 tests to see whether a Page Table Directory Entry refers to a valid Page Table.

9.1.5.6 Virtual Address Fields (VA)

The listing on the opposite page contains definitions for the fields in the CLIPPER Virtual Address (**VA**).

Lines 198 and 199 define the shift values needed to extract the Page Table Directory Index (**PTDE_PTDI**) and Page Table Index (**PTDE_PTI**) fields. The code in Example 9.11 extracts the Page Table Directory Index field from a Virtual Address.

Example 9.10

```
loadw   SPTblD,r0          # R0 = Page Table Dir Entry for Supv page 0x0
andi    $PTDE_F,r0         # Is there a valid Page Table for this entry?
brne    PageFault          # No - process Page Fault condition
```

Example 9.11

```
loadw   VAddress,r0        # R0 = CLIPPER Virtual Address
andi    $VA_PTDI,r0        # R0 = Page Table Directory Index bits
shli    $-SHF_VA_PTDI,r0   # R0 = Page Table Directory Index
```

```
0182: #
0183: # Page Table Directory Entries (PTDE).
0184: #
0185:     .set   PTDE_F,          0x00000001    # Page Fault
0186:     .set   PTDE_RSV1,       0x00000FFE    # Reserved Bits
0187:     .set   PTDE_PTO,        0xFFFFF000    # Page Table Offset
0188:
0189:     .set   SHF_PTDE_PTO,    12            # Shift value for PTE_PTO
0190:
0191: #
0192: # Virtual address bit definitions.
0193: #
0194:     .set   VA_PTDI,         0xFFC00000    # Page Table Directory Index
0195:     .set   VA_PTI,          0x003FF000    # Page Table Index
0196:     .set   VA_BYTE,         0x00000FFF    # Byte offset within page
0197:
0198:     .set   SHF_VA_PTDI,     22            # Shift value for VA_PTDI
0199:     .set   SHF_VA_PTI,      12            # Shift value for VA_PTI
```

9.2 EXAMPLE CLIPPER SYSTEM

This section describes the environment in which the example Bootstrap and Monitor are assumed to run. It also gives a brief overview of the program flow for both pieces of code.

9.2.1 I/O Configuration

The code fragments in this chapter make three assumptions about the CLIPPER System I/O configuration:

1. The system has an input device (named "System Input" or **SysIn**) that can deliver ASCII data to the CLIPPER Module one 8-bit byte at a time.
2. The system has an output device (named "System Output" or **SysOut**) that can accept ASCII data from the CLIPPER Module one 8-bit byte at a time.
3. The system has an input device (named "Binary Input" or **BinIn**) that can deliver binary data to the CLIPPER Module one 8-bit byte at a time.

The interface to these devices is through the following system primitive routines:

1. **GetByte**, which reads an 8-bit value from the System Input device **SysIn**, and returns it in register **r0**.
2. **PutByte**, which writes the 8-bit value contained in register **r0** to the System Output device **SysOut**.
3. **ReadPage**, which reads an entire 4,096 byte page from the Binary Input device **BinIn** into the virtual address contained in register **r0**.

The implementation details of these routines are not specified.

9.2.2 Memory Configuration

The code fragments in this chapter were written to run in a CLIPPER System consisting of 4K bytes of Read Only Memory containing bootstrap initialization code, 4K bytes of Random Access Memory available for bootstrap data and stack space, and 1M byte of Random Access Memory which is available for user and supervisor programs, data, and stack space. The ROM is contiguous from Boot Space address 0x00000000 to 0x00000FFF. The bootstrap RAM is contiguous from Boot Space address 0x00001000 to 0x00001FFF. The system RAM is contiguous from Main Memory 0x00000000 to 0x000FFFFF.

9.2.3 Bootstrap Program Flow

When the system is reset, the CPU begins execution at virtual address 0x00006000, which is mapped by the Hardwired Translation Lookaside Buffer (**HTLB**) to Boot Space address 0x00000000. The bootstrap program at that address contains code that:

1. Initializes the **PSW**, the **SSW**, and the Supervisor Stack Pointer.
2. Initializes the memory management hardware, setting all pages to unmapped, noncacheable. The Bootstrap code does not enable memory mapping.
3. Initializes the vector page so that every interrupt, system call, or trap results in a transfer of control to the null trap handler, **NullHandler**.
4. Reads the Monitor program from the **BinIn** device into memory starting at address 0x00008000.
5. Transfers control to the Monitor, also at address 0x00008000.

9.2.4 Monitor Program Flow

When the Monitor begins execution, both the **U** (User Mode) and **M** (Mapped Mode) bits in the System Status Word are clear; that is, the CPU is in supervisor mode and memory mapping is disabled.

The Monitor does the following:

1. Initializes its memory allocation tables.
2. Creates a Supervisor Page Table Directory, with one valid entry and 1,023 entries that generate page faults when referenced.
3. Creates one Supervisor Page Table that establishes the supervisor mode virtual memory environment described in section 9.4.7.8.
4. Creates a User Page Table Directory, with one valid entry and 1,023 entries that generate page faults when referenced.
5. Creates one User Page Table that establishes the user mode virtual memory environment described in section 9.4.10.1.
6. Resets both Instruction and Data CAMMUs and enables memory mapping.
7. Initializes the Exception Vector Table to point to the Monitor Trap, System Call, and Interrupt handler routines.
8. Enters the command prompt, input, and execute loop.

9.3 AN EXAMPLE BOOTSTRAP PROGRAM

This section contains the annotated listing of a simple bootstrap program. This code fragment illustrates the techniques you must use to initialize a CLIPPER System and begin executing a program in Supervisor Mode.

9.3.1 Bootstrap Prologue - HTLB Pages

Note the entry conditions after hardware reset:

- The vector page contains unknown data. If an interrupt, trap or system call is taken before the page is initialized, the bootstrap code will lose control of the system.
- Interrupts are disabled.
- Mapping is disabled. The CAMMUs use the Unmapped System Tag to control all memory accesses (except those mapped through the HTLB).
- The stack pointer has not been set. Do not enable interrupts or allow a trap to be taken until you initialize the stack pointer. Also, do not execute any of the following stack reference instructions:

• **call**	• **restd*n***	• **reti**
• **calls**	• **restw*n***	• **saved*n***
• **popw**	• **restur**	• **saveur**
• **pushw**	• **ret**	• **savew*n***

The code on the facing page documents the virtual pages that are accessed through the hardwired TLB.

- Virtual page 0 points to Main Memory page 0. The interrupt, trap, and system call vectors **must** be located here. The bootstrap code initializes each of the 512 Vector Table entries to point to the null interrupt handler, **NullHandler**. It also sets the associated SSW value to zero.
- Virtual page 1 points to Main Memory page 1. Virtual page 2 points to Main Memory page 2. Virtual page 3 points to Main Memory page 3. None of these pages is used by the example code in this chapter, because they are left unprotected by the HTLB. The pages could be used for supervisor code, data, or stack memory segments, but they would not be secure.
- Virtual page 4 points to I/O page 0. All of the CAMMU and TLB registers are accessed through this page. Virtual page 5 points to I/O page 1. It is available for user I/O devices, but is unused by the code in this chapter.

```
0001: #
0002: # Example Bootstrap Program.
0003: #
0004: # Entry Conditions:
0005: #   Interrupts Off
0006: #   Mapping off
0007: #   SP not set
0008: #
0009:     .text
0010: _00000000:
0011: #
0012: # Page 0 is accessed (in Supervisor Mode) through a Hardwired TLB
0013: # entry that points to Main Memory.  Page 0 always contains the
0014: # Exception Vector Table.
0015: #
0016:     .org   _00000000+0x00000000    # Virtual Pg 0 = Main Mem Pg 0
0017:     .set   VECTORS,0x00000000
0018:
0019: #
0020: # Pages 1, 2, and 3 are accessed (in Supervisor Mode) through a
0021: # Hardwired TLB # entry that points to Main Memory.  These pages are
0022: # unused by the example Bootstrap program.
0023: #
0024:     .org   _00000000+0x00001000    # Virtual Pg 1 = Main Mem Pg 1
0025:
0026:     .org   _00000000+0x00002000    # Virtual Pg 2 = Main Mem Pg 2
0027:
0028:     .org   _00000000+0x00003000    # Virtual Pg 3 = Main Mem Pg 3
0029: # Nothing in Pages 1, 2, and 3.
0030:
0031: #
0032: # Pages 4 and 5 are accessed (in Supervisor Mode) through
0033: # Hardwired TLB entries that point to I/O Space.
0034: #
0035:     .org   _00000000+0x00004000    # Virtual Pg 4 = I/O Pg 0
0036:
0037:     .org   _00000000+0x00005000    # Virtual Pg 5 = I/O Pg 1
```

9.3.2 Bootstrap Initialization

The bootstrap program is assembled with its origin at virtual address 0x00006000. The CPU begins execution here immediately after system reset or power up.

The bootstrap code loads the supervisor stack pointer with its initial value, **SSTKTOP**, which has been set to 0x00008000. Since the stack grows downward, this effectively dedicates virtual addresses from 0x00007000 to 0x00007FFF to the supervisor stack. Virtual page 0x007 is mapped by the HTLB to point to Boot Space page 0x001, so this Bootstrap RAM is used temporarily. Later, when the Monitor program establishes its virtual memory environment, it will move the supervisor stack into Main Memory.

Next, the boot program clears the **PSW** and **SSW** registers to ensure that they begin in a known state.

Next, the bootstrap initializes both the I-CAMMU and the D-CAMMU, by writing into their Control Registers. The CAMMUs are programmed as follows:

- The Unmapped System Tag bits (**CTL_UST**) are set to **UST_NCMM**, which means non cacheable Main Memory. As long as mapped mode is disabled (the **SSW_M** bit is clear in the SSW), these bits are asserted as the System Tag bits on every read and write access to virtual memory. Later, when mapping mode is enabled in the supervisor mode program (the **SSW_M** bit is set in the SSW), the **CTL_ UST** bits are ignored and the **PTE_ST** bits in each Page Table Entry are asserted on every virtual memory access.
- All of the Enable Watch bits (**CTL_EWIR**, **CTL_EWIW**, and **CTL_EWCW**) are set in the CAMMU Control Register. This ensures that all I/O and CPU accesses will retrieve the most current copy of each data word.
- The Enable Prefetch bit (**CTL_EP**) is set in the I-CAMMU only.

The bootstrap next initializes the Vector page (virtual and real page 0x000). There are 512 entries, each containing a 4-byte handler address and a 4-byte SSW value. The bootstrap sets each entry so that it points to the null interrupt handler **NullHandler**, and enters with an SSW value of zero.

Finally, the bootstrap loads the supervisor mode program (perhaps an operating system) at virtual address **PGMSTART** (0x00008000 in our example) and transfers control to it. The internals of the **LoadPgm** routine are not specified here.

```
0038: #
0039: # Pages 6 and 7 are accessed (in Supervisor Mode) through
0040: # Hardwired TLB entries that point to Bootstrap program Space.
0041: #
0042: # ************************************
0043: # *                                  *
0044: # *   Boot Space RAM Starts Here.    *
0045: # *                                  *
0046: # ************************************
0047:     .org   _00000000+0x00006000    # Virtual Pg 6 = Boot Pg 0
0048: Boot:
0049:     loadi  $SSTKTOP,sp       # Set up Supervisor stack pointer
0050:     loadq  $0,r0
0051:     movwp  r0,ssw            # Reset SSW
0052:     movwp  r0,psw            #  and PSW
0053:
0054: # Initialize both CAMMUs
0055:     loadi  $(C_CTL_UST_NCMM|C_CTL_EWIR|C_CTL_EWIW|C_CTL_EWCW|C_CTL_DP),r0
0056:     storw  r0,C_D_CTL
0057:     loadi  $(C_CTL_UST_NCMM|C_CTL_EWIR|C_CTL_EWIW|C_CTL_EWCW|C_CTL_EP),r0
0058:     storw  r0,C_I_CTL
0059:
0060: # Set up the Vector Page
0061:     loadi  $NullHandler,r0
0062:     loadq  $0,r1
0063:     loadi  $512,r6
0064:     loadi  $VECTORS,r7
0065: _vectloop:
0066:     storw  r0,(r7) # Interrupt Handler Address
0067:     storw  r1,4(r7)          # SSW for Interrupt Handler
0068:     addq   $8,r7
0069:     subq   $1,r6
0070:     brne   _vectloop
0071:
0072: # Load and execute a Supervisor Mode program.
0073:     loadi  $PGMSTART,r0
0074:     call   sp,LoadPgm
0075:     b      PGMSTART
```

9.3.3 Bootstrap Auxiliary Routines

This page of code contains support routines for the bootstrap program. The supervisor mode program also could use these routines, but it cannot determine their addresses.

PutStr() expects the address of a null-terminated string in **r0**. It transfers the address of the string into **r1** because the **PutByte()** routine expects the character in **r0**. **PutStr()** calls **PutByte()** repeatedly to write individual message characters to the **SysOut** device.

NullHandler() is the bootstrap's interrupt, trap, and system call handler. It calls **PutStr()** to output a message to the **SysOut** device, and then hangs. The boot code does not expect any interrupts or traps, but it is possible that an invalid condition will arise. The **NullHandler** routine is adequate to prevent a runaway system. Note that the SSW is zero upon entry to **NullHandler**. This value is obtained from the vector table when the **INTRAP** sequence transfers control to the interrupt or trap handler.

```
0076: #
0077: # Output a string to the system output device.
0078: # Expects:
0079: #   R0 = Address of null-terminated string.
0080: # Returns:
0081: #   Nothing
0082: #
0083:     .align 2
0084: PutStr:
0085:     movw   r0,r1
0086:     b      _pstest          # Test for null string
0087:
0088: _psloop:
0089:     call   sp,PutByte
0090:     addq   $1,r1
0091: _pstest:
0092:     loadbu (r1),r0          # Get next byte of string
0093:     cmpq   $0,r0            # Is it the terminator?
0094:     bcne   _psloop          #  No - output it
0095:     ret    sp               #  Yes - all done
0096:
0097: #
0098: # Bootstrap program Interrupt and Trap Handler.
0099: # Entry:
0100: #   Nothing
0101: # Returns:
0102: #   Does not return.
0103: #
0104:     .align 2
0105: NullHandler:
0106:     loada  _trapmsg,r0
0107:     call   sp,PutStr
0108:
0109: Halt:
0110:     movpw  ssw,r0
0111:     andi   $SSW_M,r0        # Preserve the mapping state
0112:     movwp  r0,ssw           # Interrupts off
0113:     wait   # Wait forever (except NMI)
0114:     b      Halt
0115:
0116: _trapmsg:
0117:     .byte  'N,'o,' ,'t,'r,'a,'p,' ,'h,'a,'n,'d,'l,'e,'r,'.,10,0
0118:
0119: # ********************************
0120: # *                              *
0121: # *   Boot Space ROM Ends Here.  *
0122: # *                              *
0123: # ********************************
```

Virtual Page 7 is mapped by the HTLB to point to the Bootstrap RAM. The bootstrap code places the Supervisor Stack here. Later, after establishing its virtual memory environment, the Monitor program moves the Supervisor Stack into high virtual memory. In the example hardware configuration, virtual page 7 (Boot Space page 1) contains 4K bytes of RAM. However, this need not be the case: in your own system, Boot Space pages 0 and 1 can contain either ROM or RAM or a mixture of both, as in this example.

The bootstrap code ends here. It is not meant to be exhaustive: more complete initialization is performed by the Monitor program, described in section 9.4.

```
0124: # ************************************
0125: # *                                  *
0126: # *   Boot Space RAM Starts Here.    *
0127: # *                                  *
0128: # ************************************
0129:
0130:     .org   _00000000+0x00007000    # Pg 1 of Boot Space
0131:
0132:     .space 4096                    # 4,096 bytes of stack space
0133: SSTKTOP:
0134:
0135: # ********************************
0136: # *                              *
0137: # *   Boot Space RAM Ends Here.  *
0138: # *                              *
0139: # ********************************
```

9.4 AN EXAMPLE MONITOR PROGRAM

This section contains the annotated listing of a simple Monitor program. The code receives control from the Bootstrap, establishes a virtual memory environment, reinitializes the Vector Table, and then enters the command input loop. In this loop, the Monitor prompts for input from the **SysIn** device and processes the resulting command. If the command requests the Monitor to execute a user program, it establishes a user virtual memory environment and loads a program into it from the **BinIn** device. The Monitor then initializes all of the general and floating-point registers and performs a context switch to the user program. When the user program exits (either deliberately, or through an error trap), the Monitor reclaims the user's memory and prompts for the next command.

9.4.1 Monitor Prologue

The first few pages of the program contain **.set** assembler directives that assign mnemonic names to various values used by the Monitor.

9.4.1.1 CLIPPER-Dependent Constants

The **PAGE_LEN** and **NBR_PTE** constants are fixed for each CLIPPER program. They are defined here to eliminate numeric constants from the code.

9.4.1.2 Configuration Dependent Constants

The next group of constants are implementation dependent. They are subject to change from machine to machine. Our example system uses a 10 mS clock, so **CLK_FREQ** is set to 100 ticks per second. Our example system also contains 256 pages of Main Memory (**RM_PAGES**), 2 pages of I/O Space (**IO_PAGES**), and 2 pages of Boot Space (**BS_PAGES**).

9.4.1.3 Monitor and User Protection Levels

Protection levels are defined to provide both the Monitor and the user with five page types:

1. Read/Write pages, for normal data storage.
2. Read Only pages, for global system data storage.
3. Read/Write/Execute pages, for pages accessed through the HTLB.
4. Read/Execute pages, for normal program code.
5. Inaccessible pages, for the user's protected page.

9.4.1.4 System Calls

The Monitor supports five different system calls. They allow a user program to get input from **SysIn**, send output to **SysOut**, extend his data space, and return control to the Monitor.

```
0001: #
0002: # mon.s
0003: #
0004: # An example monitor program for the CLIPPER Module.
0005: #
0006: # Entry Conditions:
0007: #   Interrupts Off.
0008: #   Mapping off.
0009: #   Supervisor Stack in page 7 (Boot Space page 1).
0010: #   I- and D-CAMMU control registers set to:
0011: #       C_CTL_UST_NCMM - noncacheable main memory.
0012: #       C_CTL_EWIR - enable watch on I/O reads.
0013: #       C_CTL_EWIW - enable watch on I/O writes.
0014: #       C_CTL_EWCW - enable watch on CPU writes.
0015: #
0016:
0017: # CLIPPER-Dependent Constants.
0018:     .set   PAGE_LEN,        4096          # 4,096 bytes per page
0019:     .set   NBR_PTE,         (PAGE_LEN/4)  # Number of PTE's per page
0020:
0021: # Configuration-Dependent Constants.
0022:     .set   CLK_FREQ,        60            # 60 hz clock tick
0023:     .set   RM_PAGES,        256           # 256 pages (1Mb) of Main Memory
0024:     .set   IO_PAGES,        2             # 2 pages (8Kb) of I/O Space
0025:     .set   BS_PAGES,        2             # 2 pages (8Kb) of Boot Space
0026:
0027: # Protection levels.
0028:     .set   PROT_RW,         PTE_PL_03     # Read/Write
0029:     .set   PROT_R,          PTE_PL_06     # Read Only
0030:     .set   PROT_RWE,        PTE_PL_07     # Read/Write/Execute
0031:     .set   PROT_RE,         PTE_PL_0C     # Read/Execute
0032:     .set   PROT_NA,         PTE_PL_0F     # No Access
0033:
0034: # System Calls
0035:     .set   SYS_EXIT,        0x00          # Exit from user program
0036:     .set   SYS_EXDS,        0x01          # Extend data segment
0037:     .set   SYS_GETB,        0x02          # Get byte from SysIn
0038:     .set   SYS_PUTB,        0x03          # Put byte to SysOut
0039:     .set   SYS_PUTS,        0x04          # Put string to SysOut
```

9.4.1.5 Page Usage and Page Table Entry Values

The example Monitor program manages a number of different types of pages. Each page type has a unique usage (**PGU_XXXX**) and Page Table Entry (**PTBL_XXXX**) value. Each page type is discussed below:

VECT, HTLB, and HTLB2: HTLB Main Memory Pages
The first three page types define the HTLB pages in main memory. Their System Tag and Protection Level bits are exactly as defined by the HTLB.

IO and BS: HTLB I/O and Boot Space Pages
The I/O and Boot Space page types define the HTLB pages in I/O and Boot Space. Their System Tag and Protection Level bits are exactly as defined by the HTLB.

PTBLD and PTBL: Page Table Directory and Page Table Pages
These pages contain Page Table Directories and Page Tables, which must be noncacheable and which must always remain in real memory when they are active.

TEXT: Monitor and User Text Pages
Text pages are cached Shared, Write-Through, because they contain both code (which is cached by the I-CAMMU) and read-only data (which is cached by the D-CAMMU). The protection level is set to Read/Execute, so that text pages cannot be written.

DATA: Monitor and User Data Pages
Data pages are cached Private, Copy-Back to take full advantage of the CAMMUs.

STK: Monitor and User Stack Pages
Stack pages are cached Private, Copy Back to take full advantage of the CAMMUs.

SGBL and UGBL: Monitor and User Global Data
The global data page contains values that are of interest both to the Monitor and the user. It is non-cacheable, to ensure that the data is always correct in memory. The Monitor's copy of global data is Read/Write. The user's copy is read-only.

RESV: Reserved Pages
Reserved pages are real memory pages that are temporarily unavailable. The Monitor uses reserved pages to control the memory allocation process when it is initializing the Monitor's virtual memory environment.

PROT: Protected Page
The user address space is initialized so that page 0 allows no accesses. This protected page helps diagnose invalid references to user virtual address 0.

INIT: Initialization Pages
The Monitor uses cache purge mode whenever it may be using a page of memory that was used previously for some other purpose. For instance, the Monitor always loads user programs and data into initialization pages. This ensures that any data in the caches from the previous usage is discarded.

NONE: Unused Pages
Unused pages generate page faults when referenced. Unused pages in Main Memory, I/O Space, and Boot Space are available for allocation.

```
0040: # Page usage values.  These numbers are used as an index into the
0041: # PgTblInfo table.  (These are quick values.)
0042:     .set   PGU_VECT,        0             # Vector page
0043:     .set   PGU_HTLB1,       1             # HTLB page 1
0044:     .set   PGU_HTLB2,       2             # HTLB pages 2 & 3
0045:     .set   PGU_IO,          3             # HTLB I/O Space page
0046:     .set   PGU_BS,          4             # HTLB Boot Space page
0047:     .set   PGU_PTBLD,       5             # Page Table Directory page
0048:     .set   PGU_PTBL,        6             # Page Table page
0049:     .set   PGU_TEXT,        7             # Text page
0050:     .set   PGU_DATA,        8             # Data page
0051:     .set   PGU_STK,         9             # Stack page
0052:     .set   PGU_SGBL,        10            # Supv global data page
0053:     .set   PGU_UGBL,        11            # User global data page
0054:     .set   PGU_RESV,        12            # Reserved page
0055:     .set   PGU_PROT,        13            # Protected page (user 0)
0056:     .set   PGU_INIT,        14            # Initialization page
0057:     .set   PGU_NONE,        15            # Unused page
0058:
0059: # Page Table Entry values.
0060:     .set   PTBL_VECT,       (PTE_ST_SHWT|PROT_RWE) # Vector page
0061:     .set   PTBL_HTLB1,      (PTE_ST_PVCB|PROT_RWE) # HTLB page 1
0062:     .set   PTBL_HTLB2,      (PTE_ST_NCMM|PROT_RWE) # HTLB pages 2 & 3
0063:     .set   PTBL_IO,         (PTE_ST_NCIO|PROT_RE)  # HTLB I/O Space
0064:     .set   PTBL_BS,         (PTE_ST_NCBS|PROT_RWE) # HTLB Boot Space
0065:     .set   PTBL_PTBLD,      (PTE_ST_NCMM|PROT_RW)  # Page Table Directory
0066:     .set   PTBL_PTBL,       (PTE_ST_NCMM|PROT_RW)  # Page Table
0067:     .set   PTBL_TEXT,       (PTE_ST_SHWT|PROT_RE)  # Supv text page
0068:     .set   PTBL_DATA,       (PTE_ST_PVCB|PROT_RW)  # Supv data page
0069:     .set   PTBL_STK,        (PTE_ST_PVCB|PROT_RW)  # Supv stack page
0070:     .set   PTBL_SGBL,       (PTE_ST_NCMM|PROT_RW)  # Supv global data
0071:     .set   PTBL_UGBL,       (PTE_ST_NCMM|PROT_R)   # User global data
0072:     .set   PTBL_RESV,       PTE_F                  # Reserved page
0073:     .set   PTBL_PROT,       (PTE_ST_NCMM|PROT_NA)  # Protected page (usr 0)
0074:     .set   PTBL_INIT,       (PTE_ST_CPRG|PROT_RW)  # Initialization page
0075:     .set   PTBL_NONE,       PTE_F                  # Invalid page
```

9.4.1.6 Fixed Real and Virtual Addresses

The example Monitor program has knowledge of a few fixed addresses. The vector page addresses are defined by the CLIPPER itself. The remaining addresses are implementation dependent.

The window page (**SVPN_WNDO**) does not always have a page of real memory assigned to it. The Monitor uses it to load the user program into any page of memory. The window page always uses cache purge mode. This ensures that any data from this page remaining in the D-CAMMU's cache will be flushed when the page is changed.

```
0076: # Real memory addresses.
0077:     .set   RMAD_TVEC,       0x00000000    # Base of trap vectors
0078:     .set   RMAD_SCVEC,      0x00000400    # Base of system call vectors
0079:     .set   RMAD_IVEC,       0x00000800    # Base of interrupt vectors
0080:     .set   RMAD_SPTD,       0x0000B000    # Supervisor Page Table Directory
0081:     .set   RMAD_SPT,        0x0000C000    # Supervisor Page Table
0082:     .set   RMAD_UPTD,       0x0000D000    # Supervisor Page Table Directory
0083:     .set   RMAD_UPT,        0x0000E000    # Supervisor Page Table
0084:
0085: # Fixed offsets to the values in the global data page.
0086:     .set   G_CLK,           0             # System clock
0087:     .set   G_NAM,           4             # System name
0088:
0089: # Fixed supervisor virtual page numbers.
0090:     .set   SVPN_WNDO,       0x010         # Window page
0091:
0092: # Fixed supervisor virtual addresses.
0093:     .set   SVAD_MERR,       0x00001000    # Memory error address
0094:     .set   SVAD_SPTD,       0x0000B000    # Supervisor Page Table Directory
0095:     .set   SVAD_SPT,        0x0000C000    # Supervisor Page Table
0096:     .set   SVAD_UPTD,       0x0000D000    # Supervisor Page Table Directory
0097:     .set   SVAD_UPT,        0x0000E000    # Supervisor Page Table
0098:     .set   SVAD_GDAT,       0x0000F000    # Global Data
0099:     .set   SVAD_GCLK,       SVAD_GDAT+G_CLK       # Global system clock
0100:     .set   SVAD_GNAM,       SVAD_GDAT+G_NAM       # Global system name
0101:     .set   SVAD_WNDO,       (SVPN_WNDO<SHF_VA_PTI) # Address of window page
0102:     .set   SVAD_STK,        0x00400000            # Top of supv stack
0103:
0104: # Fixed user virtual page numbers.
0105:     .set   UVPN_GDAT,       0x001         # Global data page
0106:     .set   UVPN_TEXT,       0x002         # Base page of text segment
0107:     .set   UVPN_DATA,       0x003         # Base page of data segment
0108:     .set   UVPN_STK,        0x3FF         # Top page of stack segment
0109:
0110: # Fixed user virtual addresses.
0111:     .set   UVAD_GDAT,       (UVPN_GDAT<SHF_VA_PTI) # Address of global data
0112:     .set   UVAD_GCLK,       UVAD_GDAT+G_CLK       # Global system clock
0113:     .set   UVAD_GNAM,       UVAD_GDAT+G_NAM       # Global system name
0114:     .set   UVAD_TEXT,       (UVPN_TEXT<SHF_VA_PTI) # Address of text segment
0115:     .set   UVAD_DATA,       (UVPN_DATA<SHF_VA_PTI) # Address of data segment
0116:     .set   UVAD_STK,        ((UVPN_STK+1)<SHF_VA_PTI) # Top of user stack
```

9.4.1.7 Initial User Register Values

The Monitor uses these constants to provide a uniform execution environment for each user program. Each of the CLIPPER's registers available to the user has an initial value defined. The fact that the user's initial **PC** value is a constant implies that the Monitor does not support alternate entry points for user programs.

The initial value for the floating-point registers is a quiet NaN, as required by the IEEE floating-point standard. The value was chosen so that the letters "BAD" are visibile in the register when it is dumped in hexadecimal.

9.4.1.8 Exception-Related Values

The Monitor initializes the **SSW** values in the vector table entries for Traps, System Calls, and Interrupts. Note that interrupts are disabled for Traps and System Calls. It is not necessary to disable interrupts for these events, but you must be careful if you leave them enabled. During the procesing of traps, system calls, and interrupts, the interrupt level is taken from the **SSW** value in the vector table. This value may or may not match the interrupt level in effect at the time the exception was taken. If you decide to enable interrupts during exception processing, you should enter the exception handler with interrupts disabled and then merge the interrupt level from the stacked **SSW** into the processor's **SSW** .

Every exception handler in the Monitor places the address of the INTRAP stack frame into register **r14**. The **INTRAP_XXX** constants define the offsets of the various registers relative to register **r14**. For example, the code in Example 9.12 retrieves the **SSW** in effect at the time the trap exception was taken and places it into register **r0**.

9.4.1.9 Message Parameter Flags

The routine **PrtMsgV** prints a message containing a variable on the **SysOut** device. The **PARM_XXXX** constants define whether the parameter is an 8-bit value, a 32-bit value, or null.

9.4.1.10 ASCII Values

Various Monitor error and informational messages use the newline and space character. They are defined mnemonically to make the code more readable.

9.4.1.11 Opcodes used as Constants

In several places, the Monitor retrieves the instruction that caused the current exception condition and compares it to expected values. The expected opcodes are defined here.

Example 9.12

```
loadw    INTRAP_SSW(r14),r0 # R0 = stacked SSW
```

```
0117: # Initial values for the user's registers
0118:     .set   INIT_U_SSW,      (SSW_U|SSW_M|SSW_ECM|SSW_EI|SSW_IL)
0119:     .set   INIT_U_PSW,      0x00000000
0120:     .set   INIT_U_PC,       UVAD_TEXT      # Initial PC address
0121:     .set   INIT_U_SP,       UVAD_STK       # Top of user stack
0122:     .set   INIT_U_REG,      0x00000000     # Registers R0-R14
0123:     .set   INIT_U_FPH,      0x7FFBAD75     # Floating Point high NaN
0124:     .set   INIT_U_FPL,      0xA0000000     # Floating Point low NaN
0125:
0126: # Default trap, system call, and interrupt SSW values.
0127:     .set   TRAP_SSW,        SSW_M          # Trap processor SSW
0128:     .set   SC_SSW,          SSW_M          # System Call processor SSW
0129:     .set   INT_SSW,         (SSW_M|SSW_EI) # Interrupt processor SSW
0130:     .set   HANG_SSW,        SSW_M          # SSW for invalid Trap, Int, SC
0131:
0132: # Offsets to the registers in the INTRAP stack frame.
0133:     .set   INTRAP_PC,       8              # Offset to PC
0134:     .set   INTRAP_SSW,      4              # Offset to SSW
0135:     .set   INTRAP_PSW,      0              # Offset to PSW
0136:
0137: # Message parameter flags.  (These are quick values.)
0138:     .set   PARM_NULL,       0x0            # No msg parameter
0139:     .set   PARM_BYTE,       0x1            # Msg parameter is a byte
0140:     .set   PARM_LONG,       0x2            # Msg parameter is a long
0141:
0142: # ASCII Values.
0143:     .set   NL,              0x0A           # New line character
0144:     .set   CR,              0x0D           # Carriage return character
0145:     .set   SP,              0x20           # Space character
0146:
0147: # Opcodes as constants
0148:     .set   OP_NOOP,         0x0000         # noop instruction
0149:     .set   OP_PUSHW,        0x1400         # pushw instruction
0150:     .set   OP_TRAPFN,       0xB43E         # trapfn instruction
0151:
0152: # Monitor Command Values.
0153:     .set   CMD_QUIT,        20             # 'q' command
0154:     .set   CMD_HELP,        21             # '?' command
```

9.4.2 Memory Management Tables

This section of Monitor code initializes the memory allocation table, the Page Table Directories and the Page Tables, and then enables virtual memory mapping.

9.4.2.1 Memory Allocation Tables

The Monitor is responsible for allocating memory in which to load user programs, and then freeing it when the program exits. The dynamic nature of memory usage requires that the Monitor keep a table documenting whether or not each real memory page in the system is in use. Since this implementation supports shared memory pages, the "in use" information must be a count of users, rather than a simple binary flag. The **RMUseCnt**, **IOUseCnt**, and **BSUseCnt** tables contain this information for each of the relevant CLIPPER address spaces. The Monitor also tracks how each page is used. The **RMUsage**, **IOUsage**, and **BSUsage** tables contain this information

The Monitor first initializes the Main Memory, I/O Space and Boot Space usage tables (**RMUsage**, **IOUsage**, and **BSUsage**). These tables contain one entry for each page of memory or space. Thus, the tables are bounded by **RM_PAGES**, **IO_PAGES**, and **BS_PAGES** respectively. Each entry contains one of the **PGU_XXXX** values defined above. The first instruction loads a copy of **PGU_NONE** into each byte of register **r2**. The three **initc** instructions each initialize one of the tables.

Next, the Monitor clears the Main Memory, I/O Space, and Boot Space using count tables (**RMUseCnt**, **IOUseCnt**, and **BSUseCnt**). Each table is bounded as above, and each table entry defines the number of users of the respective page. Every unused page in the system has a use count of zero. Every other page has a use count of one except the global data page, which has a use count of two, since both the user and the Monitor have copies of the page.

9.4.2.2 Reserved Memory

Next, the Monitor reserves Main Memory pages 4, 5, 6, and 7. These four pages are located *behind* the two HTLB I/O pages and the two HTLB Boot pages. In order to reference these pages, the Monitor must use mapped mode and must reference them at a virtual address that is not translated by the HTLB. The Monitor reserves these pages here so that it can use the normal memory allocation code when initializing its page tables. It frees these four pages after initializing its own memory and before loading the first user program. Thus, the "hidden" pages are available for user text, data, and stack pages.

```
0155: # ****************************************
0156: # *                                      *
0157: # *   Monitor text segment begins here.  *
0158: # *                                      *
0159: # ****************************************
0160:     .text
0161:     .globl Monitor
0162:     .align 2
0163: Monitor:
0164:
0165: # Initialize the Main Memory, I/O Space and Boot Space Usage Tables.
0166:     loadi  $((PGU_NONE<24)|(PGU_NONE<16)|(PGU_NONE<8)|PGU_NONE),r2
0167:     loadi  $RM_PAGES,r0
0168:     loada  RMUsage,r1
0169:     initc
0170:
0171:     loadi  $IO_PAGES,r0
0172:     loada  IOUsage,r1
0173:     initc
0174:
0175:     loadi  $BS_PAGES,r0
0176:     loada  BSUsage,r1
0177:     initc
0178:
0179: # Initialize the Main Memory, I/O Space and Boot Space Usage Count Tables.
0180:     loadq  $0,r2
0181:     loadi  $RM_PAGES,r0
0182:     loada  RMUseCnt,r1
0183:     initc
0184:
0185:     loadi  $IO_PAGES,r0
0186:     loada  IOUseCnt,r1
0187:     initc
0188:
0189:     loadi  $BS_PAGES,r0
0190:     loada  BSUseCnt,r1
0191:     initc
0192:
0193:     loadq  $4,r0
0194:     call   sp,ResvRM        # Reserve page 4
0195:     loadq  $5,r0
0196:     call   sp,ResvRM        # Reserve page 5
0197:     loadq  $6,r0
0198:     call   sp,ResvRM        # Reserve page 6
0199:     loadq  $7,r0
0200:     call   sp,ResvRM        # Reserve page 7
```

9.4.3 Virtual Memory Tables

This section of Monitor code initializes the Page Table Directories and the Page Tables, and then frees the reserved real memory.

9.4.3.1 Supervisor Page Tables

The Monitor uses the **SetGrps** routine to initialize many of its tables. **SetGrps** is a table-driven function that expects three parameters:

- The base address of the target table. This is the system table that is being initialized.
- The base address of the source table. Each entry in this table has three fields: a count field, specifying how many target table entries use the format described by this source table entry; a template field, which describes the basic target table entry format; and a modifier field, containing a value to be applied to the template field for each succeeding target table entry.
- The address of the target table processor. **SetGrps** calls this processor to initialize each group of table entries in the target table.

SetGrps is documented in detail below.

The Monitor uses the **SetGPtde** (set a group of Page Table Directory entries) function to initialize the Page Table Directories. The **SptdGrps** table describes the various groups of supervisor directory entries.

The Monitor uses the **SetGPte** (set a group of Page Table entries) function to initialize the Page Tables. The **SptGrps** table describes the various groups of supervisor Page Table entries.

9.4.3.2 User Page Tables

The Monitor again uses the **SetGPtde** and **SetGPte** functions to initialize the user Page Table Directory and Page Table. This time however, it is the **UptdGrps** and **UptGrps** tables that describe the appropriate table entries.

9.4.3.3 Releasing Reserved Memory

After initializing all of the virtual memory tables, the Monitor releases Real Memory Pages 4 through 7. This memory is marked as available by the **FreePgRM** routine, so that the Monitor can reuse it when loading user programs.

```
0201: # Initialize the Supervisor Page Table Directory.
0202:     loada  SptdGrps,r5      # Copy from address
0203:     loada  SetGPtde,r4      # Group setup processor
0204:     loada  SVAD_SPTD,r3     # Address of the Supv Page Table Directory
0205:     call   sp,SetGrps
0206:
0207: # Initialize the Supervisor Page Table.
0208:     loada  SptGrps,r5       # Copy from address
0209:     loada  SetGPte,r4       # Group setup processor
0210:     loada  SVAD_SPT,r3      # Address of the Supv Page Table Directory
0211:     call   sp,SetGrps
0212:
0213: # Initialize the Supervisor Page Table Directory Offsets.
0214:     loada  RMAD_SPTD,r0     # Addr of Supervisor Page Table Directory
0215:     storw  r0,C_I_SPDO      # Set both CAMMUs
0216:     storw  r0,C_D_SPDO
0217:
0218: # Initialize the User Page Table Directory.
0219:     loada  UptdGrps,r5      # Copy from address
0220:     loada  SetGPtde,r4      # Group setup processor
0221:     loada  SVAD_UPTD,r3     # Address of the User Page Table Directory
0222:     call   sp,SetGrps
0223:
0224: # Initialize the User Page Table.
0225:     loada  UptGrps,r5       # Copy from address
0226:     loada  SetGPte,r4       # Group setup processor
0227:     loada  SVAD_UPT,r3      # Address of the User Page Table Directory
0228:     call   sp,SetGrps
0229:
0230: # Initialize the User Page Table Directory Offsets.
0231:     loada  RMAD_UPTD,r0     # Addr of User Page Table Directory
0232:     storw  r0,C_I_UPDO      # Set both CAMMUs
0233:     storw  r0,C_D_UPDO
0234:
0235: # Free the reserved real memory
0236:     loadq  $4,r0
0237:     call   sp,FreeRM
0238:     loadq  $5,r0
0239:     call   sp,FreeRM
0240:     loadq  $6,r0
0241:     call   sp,FreeRM
0242:     loadq  $7,r0
0243:     call   sp,FreeRM
```

9.4.3.4 Enabling Virtual Memory

After initializing the Page Table Directories and Page Tables, the Monitor is ready to enable mapped mode on the CLIPPER. First, it resets both the Instruction and Data CAMMUs, to ensure that they are in a known state. Then, the Monitor sets the **SSW_M** bit in the Supervisor Status Word register. Note that the **movwp** instruction references the **SSW** register, and not the **SSWF** pseudo-register. You should always use **SSWF** with the **movwp** instruction unless you are changing the state of the **SSW_M**, **SSW_K**, **SSW_U**, or **SSW_KU** bits.

Once virtual memory is established, the Monitor relocates its stack into high virtual memory where it stays as long as the system is running.

9.4.4 Initializing The Exception Vector Table

Next, the Monitor initializes the Exception Vector Table. Again, it calls **SetGprs** to do the work, but this time using the **SetGVec** (set a group of vectors) processor. Three tables control the initialization: **TrapGrps**, which describes the trap handlers; **SCGrps**, which describes the system call handlers; and **IntGrps**, which describes the interrupt handlers. Each of these tables contains the address of the exception processor and the **SSW** to be used for the exception.

9.4.5 Final Initialization

Next, the Monitor initializes the global data page. This page is visible to both the supervisor and the user program (when it is running). It contains a copy of the system name string and current value of the system clock.

Finally, the Monitor enables interrupts. Interrupts must not be enabled before both the Exception Vector Table and the global data page are initialized, since the clock interrupts the CLIPPER every 10 mS and updates the time in the global data page.

```
0244: # Reset the CAMMUs.
0245:     loadi  $C_RS_ALL,r0
0246:     storw  r0,C_D_RS
0247:     storw  r0,C_I_RS
0248:
0249: # Turn on memory mapping.
0250: setmap:
0251:     movpw  ssw,r0
0252:     ori    $SSW_M,r0
0253:     movwp  r0,ssw
0254:
0255: # ****************************************************************
0256: # *                                                              *
0257: # *   From this point on, virtual memory mapping is enabled.     *
0258: # *                                                              *
0259: # ****************************************************************
0260:
0261: # Move the supervisor stack where it belongs.
0262:     loada  SVAD_STK,sp
0263:
0264:     pushw  r0,sp
0265: # Initialize the Trap portion of the Vector Table.
0266:     loada  TrapGrps,r5      # Copy from address
0267:     loada  SetGVec,r4       # Group setup processor
0268:     loada  RMAD_TVEC,r3     # Address of the Trap of Vector page
0269:     call   sp,SetGrps
0270:
0271: # Initialize the System Call portion of the Vector Table.
0272:     loada  SCGrps,r5        # Copy from address
0273:     loada  SetGVec,r4       # Group setup processor
0274:     loada  RMAD_SCVEC,r3    # Address of System Call portion of Vector page
0275:     call   sp,SetGrps
0276:
0277: # Initialize the Interrupt portion of the Vector Table.
0278:     loada  IntGrps,r5       # Copy from address
0279:     loada  SetGVec,r4       # Group setup processor
0280:     loada  RMAD_IVEC,r3     # Address of Interrupt portion of Vector page
0281:     call   sp,SetGrps
0282:
0283: # Initialize the Global Data page.
0284:     call   sp,InitGbl
0285:
0286: # Enable interrupts.
0287:     movpw  ssw,r0
0288:     ori    $(SSW_EI+SSW_IL),r0     # EI bit + lowest level
0289:     movwp  r0,ssw
0290:
0291: # Display the System Name.
0292:     loada  SVAD_GNAM,r0
0293:     call   sp,PutStr
```

9.4.6 The Command Loop

The monitor contains the supervisor's main execution loop. For each iteration, the supervisor:

- Reads a valid command from the **SysIn** device.
- Loads the address of the command processor routine from the command processor table (**_monptbl**).
- Transfers control to the command processor routine.

The Monitor recognizes two commands:

h Halt the Monitor program.

x (Attempt to) load and execute a user program.

The **_monptbl** table contains the address of the processor for each command.

9.4.6.1 MonHalt - Halt the Monitor Program

MonHalt prints a disgnostic message and then branches to the **Halt** routine, to shutdown the system.

```
0294: # ****************************************************************************
0295: # *                                                                          *
0296: # *   This is the main monitor loop.  On each pass, the Monitor loads        *
0297: # *   and executes a user mode program.  When the program exits (either      *
0298: # *   deliberately or as a result of an error), control returns to CSwitch   *
0299: # *   which returns to Monitor.  The Monitor then frees the memory used      *
0300: # *   by the program and repeats the entire sequence.                        *
0301: # *                                                                          *
0302: # ****************************************************************************
0303: MonLoop:
0304:     call   sp,MonCmd        # R0 = command index
0305:     shli   $2,r0            # R0 = processor table offset
0306:     loadw  _monptbl(r0),r0  # R0 = processor address
0307:     call   sp,(r0)          # Do what was asked
0308:     b      MonLoop
0309:
0310: # Monitor command processor table.
0311:     .align 4
0312: _monptbl:
0313:     .long  MonHalt          # Process 'h' command
0314:     .long  ExecUser         # Process 'x' command
0315:
0316: #
0317: # Halt the Monitor.
0318: #
0319: MonHalt:
0320:     .align 2
0321:     loada  _haltmsg,r0
0322:     call   sp,PutStr        # Print the message
0323:     b      Halt             # Halt the system
0324:
0325: _haltmsg:
0326:     .byte  '*,' ,'S,'u,'p,'e,'r,'v,'i,'s,'o,'r,' ,'H,'a,'l,'t,'e,'d,NL,0
```

9.4.6.2 ExecUser - Load and Execute a User Program

ExecUser is the processor for the Monitor's "**x**" command. It does the following:

- Attempts to load a user program from the **SysIn** device. If the load fails, **LoadUPgm** returns with the carry bit set. **ExecUser** displays a failure message and returns to the top of the loop.
- Initializes all of the user's registers and the floating-point registers. The Monitor does not use any of the floating-point registers, so they are all available to the user without saving their contents.
- Performs a context switch, putting the user code into operation in user mode. The user program runs until it either commits a fatal error or gives up the CPU by executing a **SYS_EXIT** system call. When the user program exits (for whatever reason) control eventually returns to the **CSwitch** routine, which returns to the monitor.
- Displays the run time of the user program.
- Frees all of the memory allocated to the user process.

```
0327: #
0328: # Load and Execute a user program.
0329: # Expects:
0330: #    Nothing
0331: # Returns:
0332: #    Nothing
0333: #
0334:     .align 2
0335: ExecUser:
0336:     call   sp,LoadUPgm      # Load a user program, setup its page table
0337:     bc     _exfail          # Couldn't load the program
0338:     call   sp,InitUReg      # Initialize the user's registers
0339:     loadq  $0,r0
0340:     storw  r0,U_RunTime     # Clear the run time value
0341:
0342:     call   sp,CSwitch       # Execute the user's program
0343:
0344:     loada  _erunmsg,r0      # R0 = message address
0345:     loadw  U_RunTime,r1     # R1 = parameter
0346:     loadq  $PARM_LONG,r2
0347:     call   sp,PrtMsgV       # Print the run time message
0348:     loadi  $NL,r0
0349:     call   sp,PutByte       # Print another newline
0350:     loada  SVAD_UPT,r0
0351:     call   sp,FreePRPT      # Free all pages referenced by a Page Table
0352:     ret    sp
0353:
0354: _erunmsg:
0355:     .byte  '*,' ,'R,'u,'n,'n,'i,'n,'g,' ,'T,'i,'m,'e,' ,'=,' ,0
0356:
0357: _exfail:
0358:     loada  _exfmsg,r0
0359:     call   sp,PutStr
0360:     ret    sp
0361:
0362: _exfmsg:
0363:     .byte  '*,' ,'M,'o,'n,'i,'t,'o,'r,':,' ,'F,'a,'i,'l,'e,'d
0364:     .byte  ' ,'t,'o,' ,'L,'o,'a,'d,' ,'U,'s,'e,'r
0365:     .byte  ' ,'P,'r,'o,'g,'r,'a,'m,NL,0
```

9.4.6.3 MonCmd - Get a Valid Monitor Command

MonCmd prints a prompt string on the **SysOut** device and then gets a single byte from **SysIn**. If the input is a newline character, **MonCmd** reissues the prompt and gets another input byte. Otherwise, **MonCmd** reads all remaining input up to a a newline character.

Next, **MonCmd** searches the **_mcmdtbl**, comparing the input to the legal commands. If the input matches a table entry, **MonCmd** returns the table index in register **r0**. If the input is not present in the table, **MonCmd** prints a diagnostic message and repeats the entire process. The routine does not return until the user has entered a valid command.

```
0366: #
0367: # Get a Monitor command from the SysIn device
0368: # Expects:
0369: #   Nothing
0370: # Returns:
0371: #   R0 = command index
0372: #
0373:     .align 2
0374: MonCmd:
0375:     loada  _mprompt,r0      # Print the prompt string
0376:     call   sp,PutStr
0377:     loadi  $SP,r0
0378:     call   sp,PutByte
0379:     call   sp,GetByte       # R0 = input from SysIn
0380:     cmpi   $NL,r0           # New line?
0381:     bceq   MonCmd           #  Yes - get another command
0382:     pushw  r0,sp            #  No - save input
0383: _mcflush:
0384:     call   sp,GetByte       # R0 = extra input
0385:     cmpi   $NL,r0
0386:     bcne   _mcflush         # Flush all input up to a newline
0387:     popw   sp,r1            # R1 = input
0388:     loadq  $0,r0            # R0 = command index
0389: _mcloop:
0390:     loadbu _mcmdtbl(r0),r2  # R2 = current table entry
0391:     cmpq   $0,r2            # End of table?
0392:     bceq   _mcinv           #  Yes - invalid command
0393:     cmpw   r1,r2            # Does input match current command?
0394:     bceq   _mcfound         #  Yes
0395:     addq   $1,r0            #  No - increment the index
0396:     b      _mcloop          #   and try again
0397:
0398: _mcfound:
0399:     ret    sp               # R0 = command index
0400:
0401: _mcinv:
0402:     loada  _mcimsg,r0       # R0 = message address
0403:     call   sp,PutStr        # Print it
0404:     b      MonCmd           # Get another command
0405:
0406: _mcimsg:
0407:     .byte  ' ,'I,'n,'v,'a,'l,'i,'d,' ,'C,'o,'m,'m,'a,'n,'d,NL,0
0408:
0409: _mprompt:
0410:     .byte  'M,'o,'n,'i,'t,'o,'r,'>,SP,0
0411:
0412: _mcmdtbl:
0413:     .byte  'h               # Halt the Monitor
0414:     .byte  'x               # Execute a user program
0415:     .byte  0                # End of table
```

9.4.6.4 CSwitch - Perform Context Switch

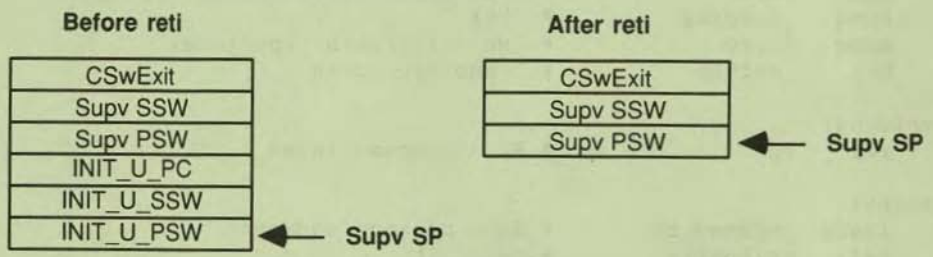
CSwitch constructs two dummy INTRAP stack frames on the supervisor stack and uses them to perform a context switch and, eventually, a return. The first dummy stack frame allows the **SYS_EXIT** handler to return to **CSwExit** when the user program terminates. The second INTRAP stack frame allows **CSwitch** to transfer control to the user program in user mode.

After constructing the two dummy stack frames, **CSwitch** clears the **U_RunTime** variable and then clears all user valid bits (**C_RS_RUV**) from both CAMMUs. This causes the CAMMUs to discard data associated with any user virtual addresses.

CSwitch uses the **reti** instruction to perform the context switch. You can think of **reti** as an instruction that allows you to set the **PSW**, **SSW** and **PC** registers at the same time. If you simply set the **SSW_U** bit directly in the status word, your program will be executing in user mode at the address after the **movwp** instruction. But the CAMMU uses the **UPDO** registers instead of the **SPDO** registers in user mode. Thus, when you set the **SSW_U** bit, you will immediately begin executing in user virtual memory, not supervisor memory. You must use the **reti** instruction to set *both* the user mode bit and the user's execution address at the same time.

Figure 9.1 illustrates the supervisor stack immediately before and after the **reti** instruction. Remember however, that the processor is executing in user mode with the user register set at address **INIT_U_PC**. The instruction after the **reti** is in supervisor virtual memory: later, the user exit processor will execute another **reti** instruction which uses the remaining dummy INTRAP stack frame to return control to **CSwExit**.

Figure 9.1 Supervisor Stack in CSwitch



```
0416: #
0417: # Context switch to the user.  Control returns to the caller of this
0418: # routine when the user program exits (either deliberately, or because
0419: # of an error).
0420: # Expects:
0421: #   Nothing
0422: # Returns:
0423: #   Nothing
0424: #
0425:     .align 2
0426: CSwitch:
0427: # Build a dummy INTRAP stack frame to return here after the user exits
0428:     loada  CSwExit,r0       # Return address
0429:     pushw  r0,sp
0430:     movpw  ssw,r0           # Get current SSW
0431:     pushw  r0,sp
0432:     movpw  psw,r0           # Get current PSW
0433:     pushw  r0,sp
0434:     storw  sp,CSwitchSP     # Save the SP for the user exit processor
0435:
0436: # Build a dummy INTRAP stack frame to transfer to the user in user mode
0437:     loadi  $INIT_U_PC,r0    # Entry point to user's program
0438:     pushw  r0,sp
0439:     loadi  $INIT_U_SSW,r0   # Initial user mode SSW
0440:     pushw  r0,sp
0441:     loadi  $INIT_U_PSW,r0   # Initial user mode PSW
0442:     pushw  r0,sp
0443:
0444: # Reset all of the user valid bits in both CAMMUs.
0445:     loadi  $C_RS_RUV,r0
0446:     storw  r0,C_D_RS        # Reset user valid lines in D-CAMMU
0447:     storw  r0,C_I_RS        # Reset user valid lines in I-CAMMU
0448:
0449: # Transfer to the user program in user mode
0450: Transfer:
0451:     reti   sp
0452:
0453: # ************************************************************
0454: # *                                                          *
0455: # *   Control returns here after the user program exits.     *
0456: # *                                                          *
0457: # ************************************************************
0458:     .align 2
0459: CSwExit:
0460:     ret    sp
```

9.4.7 Table Initialization Routines

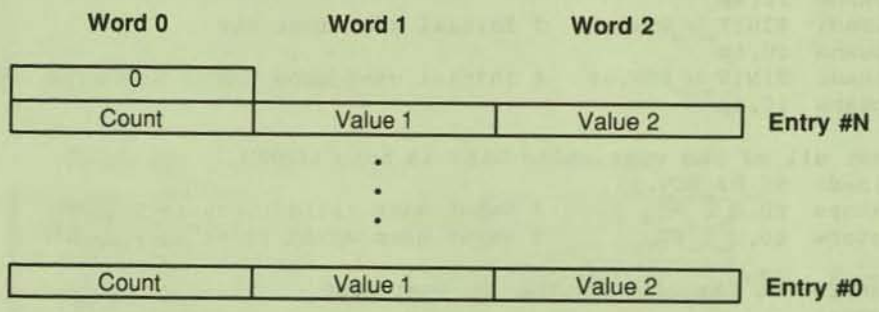
The routines in this section of code initialize various system tables, including the supervisor and user Page Table Directories and Page Tables, and the Vector Table.

9.4.7.1 SetGrps - the Master Table Initializer

The Monitor uses the **SetGrps** routine to initialize many of its tables. Figure 9.2 illustrates the format of the table that **SetGrps** expects. Each table entry contains three longwords. The first word is the number of target table entries this master table entry describes. The remaining two words contain initialization information. The end of the table is signalled by an entry with a count field of zero.

Each time it is called, **SetGrps** loads the **Count**, **Value 1**, and **Value 2** fields into registers and calls the group initializer to process the group. **SetGrps** pays no attention to the contents of the **Value 1** and **Value 2** fields. It simply passes them as parameters to the group initializer. **SetGrps** returns to its caller when it encounters a table entry with a count of zero. For safety, **SetGrps** tests the very first entry for a zero count before calling the group initializer.

Figure 9.2 Master Initializer Table Format



```
0461: # ************************
0462: # *                      *
0463: # *   Support routines   *
0464: # *                      *
0465: # ************************
0466:
0467: #
0468: # Setup groups of common entries in a system table.
0469: # Expects:
0470: #   R5 = Base address of group table
0471: #   R4 = Address of group initializer
0472: #   R3 = Base address of System table to initialize
0473: # Returns:
0474: #   Nothing
0475: #
0476:     .align 2
477: SetGrps:
0478:     b       _sgtest          # Test for empty group list
0479:
0480: _sgloop:
0481:     loadw   4(r5),r1         # R1 = Value 1
0482:     loadw   8(r5),r0         # R0 = Value 2
0483:     pushw   r5,sp
0484:     pushw   r4,sp
0485:     call    sp,(r4)          # Process this group
0486:     popw    sp,r4
0487:     popw    sp,r5
0488:     addq    $12,r5
0489: _sgtest:
0490:     loadw   (r5),r2          # R2 = count for next group
0491:     cmpq    $0,r2            # Count = 0?
0492:     bcne    _sgloop          #  No - do next group
0493:     ret     sp
```

9.4.7.2 SetGPtde - the Page Table Directory Initializer

The **SetGrps** routine calls **SetGPtde** to initialize one group of similar Page Table Directory entries. It interprets the **Value 1** and **Value 2** parameters from **SetGrps** as follows:

- **Value 1** (in register **r1**) is the Page Table Directory entry template.
- **Value 2** (in register **r0**) is zero and is ignored. Remember that **SetGrps** is called for several different types of tables. **Value 2** is required for some tables and not for others.

SetGPtde simply copies the Page Table Directory entry into the table, increments the table address, decrements the entry count, and loops until the count goes to zero. The routine does not need to check for a zero count upon entry, since **SetGrps** returns without calling the group initializer when it detects a count of zero.

9.4.7.3 SptdGrps - Supervisor Page Table Directory Groups

The **SetGrps** routine uses the **SetGPtde** routine and the **SptdGrps** table to initialize the supervisor Page Table Directory. The table contains two entries. The first entry describes one valid Page Table whose address is **RMAD_SPT** (the real memory address of the supervisor Page Table). Note that the Page Table Directory must reference the real memory address of the Page Table(s), not their virtual memory addresses. The second table entry describes 1,023 Page Table Directory entries that generate page faults. Every Page Table can address 1,024 pages of real memory. Since each page is 4,096 bytes long, each Page Table can address 4M bytes of memory. Every Page Table Directory can address 1,024 individual Page Tables. Since each Page Table can address 4M bytes of memory, the Page Table Directory can address 4G bytes of memory.

9.4.7.4 UptdGrps - User Page Table Directory Groups

The **SetGrps** routine uses the **SetGPtde** routine and the **UptdGrps** table to initialize the user's Page Table Directory. The table contains one entry that describes 1,024 Page Table Directory entries that generate page faults. Since there is no user loaded when the table is initialized, the user's virtual memory environment is empty.

```
0494: #
0495: # Set a contiguous group of Page
Table Directory Entries.
0496: # Expects:
0497: #   R3 = Start address in Page
Table Directory
0498: #   R2 = Count (Must be greater
than 0)
0499: #   R1 = PTD Entry template
0500: #   R0 = 0 - ignored
0501: # Returns:
0502: #   Nothing
0503: #
0504:     .align 2
0505: SetGPtde:
0506:     storw  r1,(r3)          # Set one Page Table Directory Entry
0507:     addq   $4,r3            # Increment the Page Table Directory address
0508:     subq   $1,r2            # Decrement the count
0509:     brne   SetGPtde         #  and loop if not zero
0510:     ret    sp
0511:
0512: #
0513: # Supervisor Page Table Directory groups for initialization.
0514: #
0515:     .align 4
0516: SptdGrps:
0517: # First entry points to Supervisor Page Table.
0518:     .long  1,               RMAD_SPT,     0      # Supervisor Page Table
0519: # Remaining entries cause page faults.
0520:     .long  1023,            PTDE_F,0
0521: # End of table.
0522:     .long  0
0523:
0524: #
0525: # User Page Table Directory groups for initialization.
0526: #
0527:     .align 4
0528: UptdGrps:
0529: # First entry points to User Page Table.
0530:     .long  1,               RMAD_UPT,     0      # User Page Table
0531: # Remaining entries cause page faults.
0532:     .long  1023,            PTDE_F,0
0533: # End of table.
0534:     .long  0
```

9.4.7.5 SetGPte - the Page Table Initializer

The **SetGrps** routine calls **SetGPte** to initialize one group of similar Page Table entries. It interprets the **Value 1** and **Value 2** parameters from **SetGrps** as follows:

- **Value 1** (in register **r1**) is zero and is ignored.
- **Value 2** (in register **r0**) is the page usage value (**PGU_XXXX**).

SetGPte is somewhat more complex than **SetGPtde**. First, **SetGPte** translates the page usage value into a Page Table Entry value (**PTBL_XXXX**) using the **PgTblInfo** table. **SetGPte** retains both values for later use and then enters a loop, which has one iteration for each Page Table entry with the same usage (that is, the **Count** field from the **SptGrps** table is the loop control variable). For each loop iteration (and hence, each Page Table entry with the same usage), **SetGPte** calls **SgptGet** to allocate a page of real memory, I/O Space or Boot Space if the page usage requires it. It then converts the real page number returned by **SptgGet** into a real address and merges it with the System Tag and Protection Level bits to create a complete Page Table entry. Finally, **SetGPte** stores the new entry, increments the Page Table address, decrements the count, and loops until the count reaches zero.

If **SgptGet** ever fails to allocate a page in the required address space, it sets the carry bit in the **PSW** and returns. When **SetGPte** detects a failure of this sort, it sends an error message to the **SysOut** device and crashes the system. Thus, there must be enough of the required memory in order for the system to run.

```
0535: # Set a contiguous group of Page Table Entries.
0536: # Expects:
0537: #    R3 = Start address in Page Table
0538: #    R2 = Count (Must be greater than 0)
0539: #    R0 = Page usage value
0540: # Returns:
0541: #    Nothing
0542:      .align 2
0543: SetGPte:
0544:      movw   r0,r1              # R1 = page usage
0545:      shli   $2,r1              # R1 = table offset
0546:      loadw  PgTblInfo(r1),r1 # R1 = System tag and protection level
0547:      movw   r0,r4              # R4 = saved usage
0548: _sgploop:
0549:      call   sp,SgptGet         # R0 = real page number
0550:      bc     _sgpfail           # Could not allocate a page
0551:      shli   $SHF_PTE_RA,r0     # R0 = real address
0552:      orw    r1,r0              # R0 = Page Table Entry
0553:      storw  r0,(r3)            # Set one Page Table Directory Entry
0554:      addq   $4,r3              # Increment the Page Table Directory address
0555:      subq   $1,r2              # Decrement the count
0556:      brne   _sgploop           #  and loop if not zero
0557:      ret    sp
0558:
0559: _sgpfail:
0560:      loada  _sgpmsg,r0         # R0 = message address
0561:      movw   r3,r1              # R1 = table address
0562:      loadq  $PARM_LONG,r2      # R2 = parm type
0563:      b      Crash              # Crash the system
0564:
0565: _sgpmsg:
0566:      .byte  '*,' ,'S,'e,'t,'G,'p,'t,'e,':,' ,'N,'o,' ,'P,'a,'g,'e,' ,'-
0567:      .byte  ' ,'T,'a,'b,'l,'e,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
```

9.4.7.6 PgTblInfo - Page Usage to Page Table Entry Translation

SetGPte uses the **PgTblInfo** table to translate a page usage value (**PGU_XXXX**) into a Page Table entry value (**PTBL_XXXX**). The page usage value identifies the type of page in question. The Page Table entry value is made up of the System Tag (**PTE_ST**) and Protection Level (PTE_PL) bits of the Page Table entry. The translation is straightforward: simply multiply the page usage value by four (shift left by two) and use the result as an index into **PgTblInfo**.

9.4.7.7 SgptGet - Allocate a Page for SetGpte

SgptGet checks to see if the new page usage is **PGU_NONE**. If it is, there is no need to allocate a page, so the routine clears the carry bit and returns. Otherwise, it saves the registers, moves the new page usage into register **r0**, and calls **AllocPage** to allocate a page of Main Memory, I/O Space, or Boot Space.

```
0568:     .align 4
0569: PgTblInfo:
0570:     .long  PTBL_VECT        # PGU_VECT - Vector page
0571:     .long  PTBL_HTLB1       # PGU_HTLB1 - HTLB page 1
0572:     .long  PTBL_HTLB2       # PGU_HTLB2 - HTLB pages 2 & 3
0573:     .long  PTBL_IO          # PGU_IO - HTLB I/O Space
0574:     .long  PTBL_BS          # PGU_BS - HTLB Boot Space
0575:     .long  PTBL_PTBLD       # PGU_PTBLD - Page Table Dir
0576:     .long  PTBL_PTBL        # PGU_PTBL - Page Table
0577:     .long  PTBL_TEXT        # PGU_TEXT - Supv text page
0578:     .long  PTBL_DATA        # PGU_DATA - Supv data page
0579:     .long  PTBL_STK         # PGU_STK - Supv stack page
0580:     .long  PTBL_SGBL        # PGU_SGBL - Supv global data page
0581:     .long  PTBL_UGBL        # PGU_UGBL - User global data page
0582:     .long  PTBL_RESV        # PGU_RESV - Reserved page
0583:     .long  PTBL_PROT        # PGU_PROT - Protected page
0584:     .long  PTBL_INIT        # PGU_INIT - Initialization page
0585:     .long  PTBL_NONE        # PGU_NONE - Invalid page
0586:
0587: #
0588: # Get a page for SetGPte.
0589: # Expects:
0590: #   R4 - new page usage
0591: # Returns:
0592: #   PSW_C = 0, R0 = real page number
0593: #   PSW_C = 1, failure
0594: #   R1,R2,R3,R4 preserved
0595: #
0596:     .align 2
0597: SgptGet:
0598:     cmpq   $PGU_NONE,r4     # New page unused?
0599:     bceq   _sggunus         #  Yes
0600:     pushw  r1,sp
0601:     pushw  r2,sp
0602:     pushw  r3,sp
0603:     pushw  r4,sp
0604:     movw   r4,r0            # R0 = new page usage
0605:     call   sp,AllocPage     # R0 = real page number
0606:     popw   sp,r4
0607:     popw   sp,r3
0608:     popw   sp,r2
0609:     popw   sp,r1
0610:     ret    sp               # PSW_C set, R0 = page number
0611:
0612: _sggunus:
0613:     loadq  $0,r0            # Page number = 0
0614:     movwp  r0,psw           # PSW_C = 0
0615:     ret    sp
```

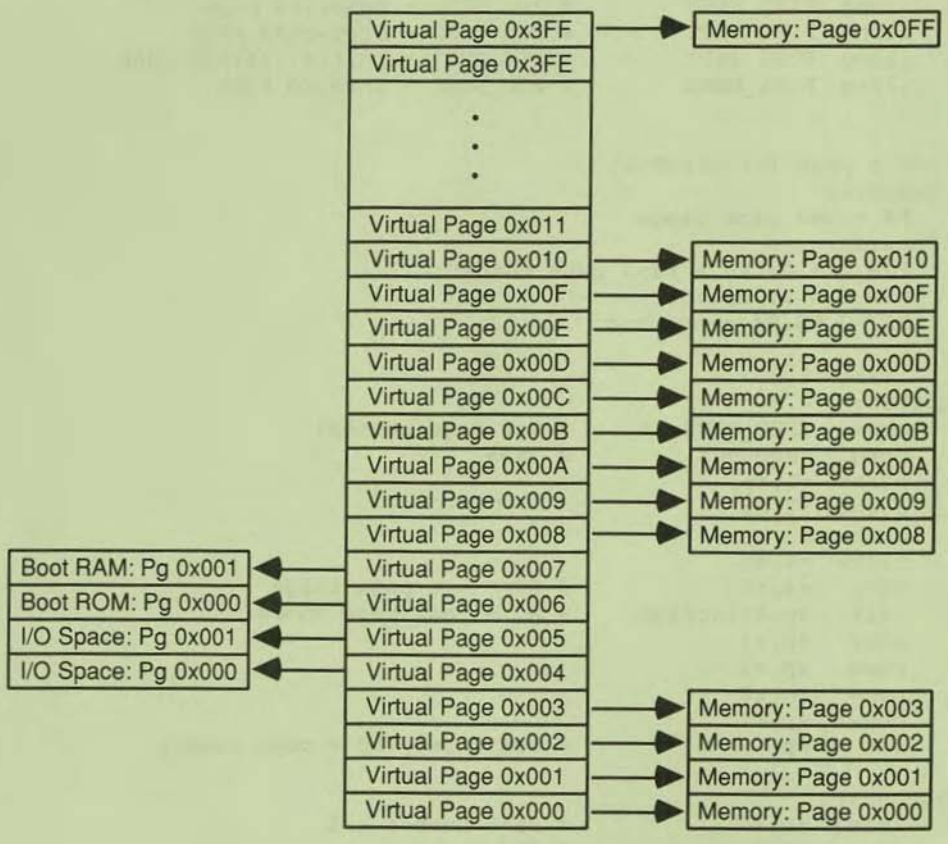
9.4.7.8 SptGrps - Supervisor Page Table Groups

The **SetGrps** routine uses the **SetGPte** routine and the **SptGrps** table to initialize the supervisor Page Table. The table contains the entries required to establish the virtual memory map shown in Figure 9.3.

9.4.7.9 UptGrps - User Page Table Groups

The **SetGrps** routine uses the **SetGPte** routine and the **UptGrps** table to initialize the user's Page Table. The table contains one entry that describes 1,024 Page Table entries that generate page faults. Since there is no user loaded when the table is initialized, the user's virtual memory environment is empty.

Figure 9.3 Monitor Virtual Memory Map



```
0616: #
0617: # Supervisor Page Table groups for initialization.
0618: #
0619:     .align 4
0620: SptGrps:
0621: # HTLB Virtual Page 0x000; Vectors.
0622:     .long  1,                0,     PGU_VECT
0623: # HTLB Virtual Page 0x001; Unused.
0624:     .long  1,                0,     PGU_HTLB1
0625: # HTLB Virtual Pages 0x002-0x003; Unused.
0626:     .long  2,                0,     PGU_HTLB2
0627: # HTLB Virtual Pages 0x004-0x005; I/O Space.
0628:     .long  2,                0,     PGU_IO
0629: # HTLB Virtual Pages 0x006-0x007; Boot Space.
0630:     .long  2,                0,     PGU_BS
0631: # Virtual Pages 0x008-0x009; Supervisor text segment.
0632:     .long  2,                0,     PGU_TEXT
0633: # Virtual Page 0x00A; Supervisor data segment.
0634:     .long  1,                0,     PGU_DATA
0635: # Virtual Page 0x00B; Supervisor Page Table Directory
0636:     .long  1,                0,     PGU_PTBLD
0637: # Virtual Page 0x00C; Supervisor Page Table
0638:     .long  1,                0,     PGU_PTBL
0639: # Virtual Page 0x00D; User Page Table Directory
0640:     .long  1,                0,     PGU_PTBLD
0641: # Virtual Page 0x00E; User Page Table
0642:     .long  1,                0,     PGU_PTBL
0643: # Virtual Page 0x00F; Global Data.
0644:     .long  1,                0,     PGU_SGBL
0645: # Virtual Page 0x010; Window page.  Initially, an invalid page.
0646:     .long  1,                0,     PGU_NONE
0647: # Virtual Pages 0x011-0x3FE; Invalid pages.
0648:     .long  1006,             0,     PGU_NONE
0649: # Virtual Page 0x3FF; Supervisor stack.
0650:     .long  1,                0,     PGU_STK
0651: # End of table.
0652:     .long  0
0653:
0654: #
0655: # User Page Table groups for initialization.
0656: #
0657:     .align 4
0658: UptGrps:
0659: # Everything is invalid to begin with.
0660:     .long  1024,             0,     PGU_NONE
0661: # End of table.
0662:     .long  0
```

Figure 9.4 illustrates the Monitor's virtual memory environment after initialization. Note that virtual pages 1, 2, and 3 are unused. They are mapped through the HTLB, which provides no protection for the pages, so the Monitor ignores them. Also note that there are two pages dedicated to Monitor code, and only one page to data. Finally, note that the supervisor stack is located between virtual address 0x003FF000 and 0x003FFFFF. Isolating the stack in this manner ensures that a stack underflow or overflow condition will generate a page fault. Actually, this will cause a double fault, since INTRAP will fail in its attempt to create a stack frame when the first fault is signalled. After initialization, the window page has been allocated at virtual address 0x00010000, but it is not mapped to point to real memory. The Monitor uses the **SetWindow** and **ClrWindow** routines, described below, to reference any real memory page in the system.

Figure 9.4 Monitor Virtual Memory Usage

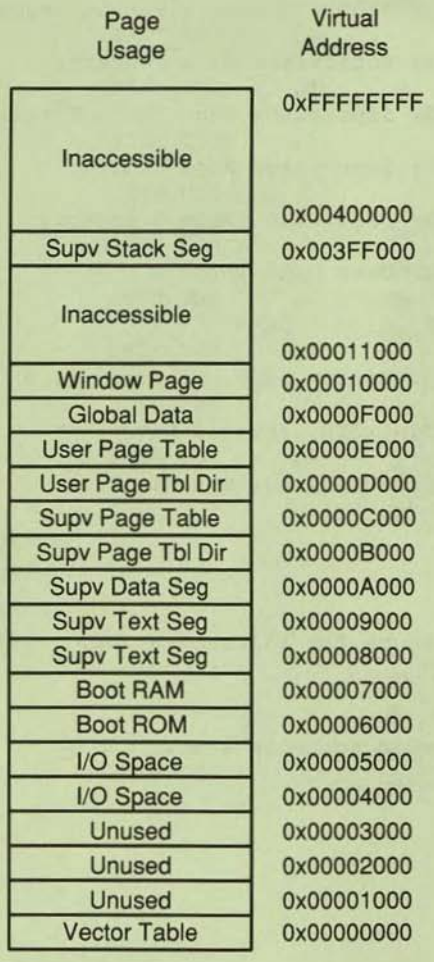
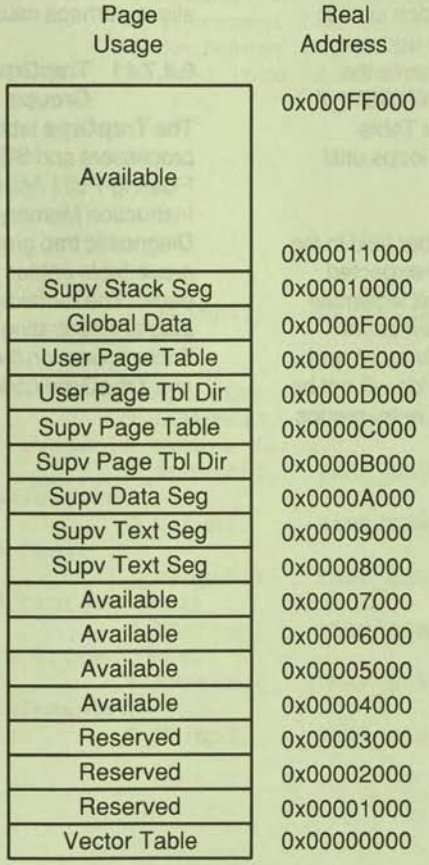


Figure 9.5 illustrates the real memory usage immediately after initialization. Note that real pages 4, 5, 6, and 7 are available. These pages are "hidden" behind the HTLB I/O and Boot Space pages. The Monitor reserves these real memory pages with the **ResvRM** routines before it calls **SetGrps** to initialize its Page Table. Notice also that all of the real addresses are identical to the virtual address, except for the page containing the supervisor's stack. It is not necessary for this to be the case, it simply makes it easier for the Monitor to go from an unmapped to a mapped environment at initialization time. In this implementation, except for these low memory pages, it is entirely coincidental when virtual and real addresses match.

Figure 9.5 Initial Real Memory Usage



9.4.7.10 SetGVec - the Vector Table Initializer

The **SetGrps** routine uses the **SetGVec** routine and the **TrapGrps**, **SCGrps**, and **IntGrps** tables to initialize the Vector Table. The reason for separating the Trap, System Call, and Interrupt vectors will become apparent in a moment.

SetGVec expects the number of entries to initialize (the **Count** parameter from the table) in register **r2**, the address of the exception processor (**Value 1** from the table) in register **r1**, and the **SSW** for the exception processor (**Value 2** from the table) in register **r0**. For each similar entry, **SetGVec** stores the processor address and **SSW** in the Vector Table, increments the interrupt level/number field in the **SSW** (**SSW_IL** and **SSW_IN**), increments the Vector Table address, decrements the count, and loops until zero.

Incrementing the interrupt level/number field in the **SSW** is a trick that helps diagnose unexpected exceptions. In this implementation, all legitimate exceptions have unique handlers: their count parameters in the group tables are always 1 and thus, their Interrupt Level/Number fields will not be incremented. The only Vector Table entry groups that have a count greater than one are the unimplemented Traps, unsupported System Calls, and unexpected Interrupts. The **SSW** for these groups always has the **SSW_EI** bit cleared, so the **SSW_IL** and **SSW_IN** fields can contain any value. If this value is different for each unexpected exception, then the exception processor can use the value to determine which invalid exception occurred. This scheme only works when the **SSW** disables interrupts upon entry to the exception processor. If interrupts were enabled, the CPU would use the altered **SSW_IL** field to determine which interrupts to allow, perhaps causing chaos in the system.

9.4.7.11 TrapGrps - Vector Table Trap Groups

The **TrapGrps** table describes the exception processors and **SSW**s for all Data Memory, Floating-Point Arithmetic, Integer Arithmetic, Instruction Memory, Illegal Operation, and Diagnostic trap groups. The Data Memory trap group table entries are shown on the opposite page. The remaining **TrapGrps** entries and a diagram illustrating the Trap portion of the Vector Table appear on the pages following the **SetGVec** and **TrapGrps** code.

```
0663: #
0664: # Set a contiguous group of vectors in the vector table
0665: # Expects:
0666: #   R3 = Start address in Vector Table
0667: #   R2 = Count (Must be greater than 0)
0668: #   R1 = Exception Processor address
0669: #   R0 = SSW value
0670: # Returns:
0671: #   Nothing
0672:     .align 2
0673: SetGVec:
0674:     storw  r1,(r3)          # Set one processor address
0675:     storw  r0,4(r3)         # Set one vector SSW
0676:     addq   $1,r0            # Modify the Interrupt Level/Number in SSW
0677:     addq   $8,r3            # Increment the Vector Table address
0678:     subq   $1,r2            # Decrement the count
0679:     brne   SetGVec          #  and loop if not zero
0680:     ret    sp
0681:
0682: #
0683: # Table of Trap vector groups.
0684: #
0685:     .align 4
0686: TrapGrps:
0687: # 0x00000000 - Unimplemented
0688:     .long  33,              T_Unimpl,     HANG_SSW+0x00
0689:
0690: # Data Memory Trap Group.
0691: # 0x00000108 - Corrected Memory Error
0692:     .long  1,               T_DCMemErr,   TRAP_SSW
0693: # 0x00000110 - Uncorrectable Memory Error
0694:     .long  1,               T_DUMemErr,   TRAP_SSW
0695: # 0x00000118 - Unimplemented
0696:     .long  2,               T_Unimpl,     HANG_SSW+0x23
0697: # 0x00000128 - Page Fault
0698:     .long  1,               T_DPageFlt,   TRAP_SSW
0699: # 0x00000130 - Read Protect Fault
0700:     .long  1,               T_DRdProt,    TRAP_SSW
0701: # 0x00000138 - Write Protect Fault
0702:     .long  1,               T_DWrtProt,   TRAP_SSW
0703: # 0x00000140 - Unimplemented
0704:     .long  8,               T_Unimpl,     HANG_SSW+0x28
```

Figure 9.6 Vector Table—Traps

Processor Address	SSW Value	Real Address
T_Unimpl	0x0400007F	0x000003F8
T_Unimpl	0x04000071	0x00000388
T_Trace	0x04000000	0x00000380
T_Unimpl	0x0400006F	0x00000378
T_Unimpl	0x04000062	0x00000310
T_PrivIns	0x04000000	0x00000308
T_IllOp	0x04000000	0x00000300
T_Unimpl	0x0400005F	0x000002F8
T_Unimpl	0x04000057	0x000002B8
T_IExProt	0x04000000	0x000002B0
T_IPageFlt	0x04000000	0x000002A8
T_Unimpl	0x04000054	0x000002A0
T_Unimpl	0x04000053	0x00000298
T_IUMemErr	0x04000000	0x00000290
T_ICMemErr	0x04000000	0x00000288
T_Unimpl	0x04000050	0x00000280
T_Unimpl	0x04000042	0x00000210
T_IDiv0	0x04000000	0x00000208
T_Unimpl	0x04000040	0x00000200
T_Unimpl	0x04000039	0x000001C8
T_FPInvOp	0x04000000	0x000001C0
T_Unimpl	0x04000037	0x000001B8
T_Unimpl	0x04000036	0x000001B0
T_Unimpl	0x04000035	0x000001A8
T_FPOflow	0x04000000	0x000001A0
T_Unimpl	0x04000033	0x00000198
T_FPDiv0	0x04000000	0x00000190
T_FPUflow	0x04000000	0x00000188
T_FPInex	0x04000000	0x00000180
T_Unimpl	0x0400002F	0x00000178
T_Unimpl	0x04000028	0x00000140
T_DWrtProt	0x04000000	0x00000138
T_DRdProt	0x04000000	0x00000130
T_DPageFlt	0x04000000	0x00000128
T_Unimpl	0x04000024	0x00000120
T_Unimpl	0x04000023	0x00000118
T_DUMemErr	0x04000000	0x00000110
T_DCMemErr	0x04000000	0x00000108
T_Unimpl	0x04000020	0x00000100
T_Unimpl	0x04000000	0x00000000

```
0705: # Floating Point Arithmetic Trap Group
0706: # 0x00000180 - Floating Inexact
0707:     .long  1,                T_FPInex,     TRAP_SSW
0708: # 0x00000188 - Floating Underflow
0709:     .long  1,                T_FPUflow,    TRAP_SSW
0710: # 0x00000190 - Floating Divide by 0
0711:     .long  1,                T_FPDiv0,     TRAP_SSW
0712: # 0x00000198 - Unimplemented
0713:     .long  1,                T_Unimpl,     HANG_SSW+0x33
0714: # 0x000001A0 - Floating Overflow
0715:     .long  1,                T_FPOflow,    TRAP_SSW
0716: # 0x000001A8 - Unimplemented
0717:     .long  3,                T_Unimpl,     HANG_SSW+0x35
0718: # 0x000001C0 - Floating Invalid Op
0719:     .long  1,                T_FPInvOp,    TRAP_SSW
0720: # 0x000001C8 - Unimplemented
0721:     .long  8,                T_Unimpl,     HANG_SSW+0x39
0722:
0723: # Integer Arithmetic Trap Group
0724: # 0x00000208 - Integer Divide by 0
0725:     .long  1,                T_IDiv0,      TRAP_SSW
0726: # 0x00000210 - Unimplemented
0727:     .long  15,               T_Unimpl,     HANG_SSW+0x42
0728:
0729: # Instruction Memory Trap Group
0730: # 0x00000288 - Corrected Memory Error
0731:     .long  1,                T_ICMemErr,   TRAP_SSW
0732: # 0x00000290 - Uncorrectable Memory Error
0733:     .long  1,                T_IUMemErr,   TRAP_SSW
0734: # 0x00000298 - Unimplemented
0735:     .long  2,                T_Unimpl,     HANG_SSW+0x53
0736: # 0x000002A8 - Page Fault
0737:     .long  1,                T_IPageFlt,   TRAP_SSW
0738: # 0x000002B0 - Execute Protect Fault
0739:     .long  1,                T_IExProt,    TRAP_SSW
0740: # 0x000002B8 - Unimplemented
0741:     .long  9,                T_Unimpl,     HANG_SSW+0x57
0742:
0743: # Illegal Operation Trap Group
0744: # 0x00000300 - Illegal Operation
0745:     .long  1,                T_IllOp,      TRAP_SSW
0746: # 0x00000308 - Privileged Instruction
0747:     .long  1,                T_PrivIns,    TRAP_SSW
0748: # 0x00000310 - Unimplemented
0749:     .long  14,               T_Unimpl,     HANG_SSW+0x62
0750:
0751: # Diagnostic Trap Group
0752: # 0x00000380 - Trace trap
0753:     .long  1,                T_Trace,      TRAP_SSW
0754: # 0x00000388 - Unimplemented
0755:     .long  15,               T_Unimpl,     HANG_SSW+0x71
0756: # End of table.
0757:     .long  0
```

9.4.7.12 SCGrps - Vector Table System Call Groups.

The **SCGrps** table describes the exception processors and **SSW**s for all System Calls. Figure 9.7 illustrates the System Call portion of the Vector Table:

9.4.7.13 IntGrps - Vector Table Interrupt Groups.

The **IntGrps** table describes the exception processors and **SSW**s for all interrupts. Figure 9.8 illustrates the interrupt portion of the Vector Table. Note that the **SSW_EI** bit is set for valid interrupts but clear for unexpected interrupts. If you elect to use the **SSW_IL** and **SSW_IN** fields to diagnose unexpected interrupts, you must not set the **SSW_EI** bit.

Figure 9.7 Vector Table - System Calls

Processor Address	SSW Value	Real Address
SC_Unsupp	0x0400007F	0x000007F8
SC_Unsupp	0x04000005	0x00000428
SC_PutS	0x04000000	0x00000420
SC_PutB	0x04000000	0x00000418
SC_GetB	0x04000000	0x00000410
SC_ExDS	0x04000000	0x00000408
SC_Exit	0x04000000	0x00000400

Figure 9.8 Vector Table - Interrupts

Processor Address	SSW Value	Real Address
I_Unexp	0x0400007F	0x00000FF8
I_Unexp	0x04000003	0x00000810
I_Clock	0x04000100	0x00000808
I_Nmi	0x04000100	0x00000800

```
0758: #
0759: # Table of Supervisor Call groups.
0760: #
0761:     .align 4
0762: SCGrps:
0763: # 0x00000400 - System Call 0x00
0764:     .long  1,              SC_Exit,      SC_SSW # SYS_EXIT call
0765: # 0x00000408 - System Call 0x01
0766:     .long  1,              SC_ExDS,      SC_SSW # SYS_EXDS call
0767: # 0x00000410 - System Call 0x02
0768:     .long  1,              SC_GetB,      SC_SSW # SYS_GETB call
0769: # 0x00000418 - System Call 0x03
0770:     .long  1,              SC_PutB,      SC_SSW # SYS_PUTB call
0771: # 0x00000420 - System Call 0x04
0772:     .long  1,              SC_PutS,      SC_SSW # SYS_PUTS call
0773: # 0x00000428 - Unsupported System Calls
0774:     .long  123,            SC_Unsupp,    HANG_SSW+0x03
0775: # End of table.
0776:     .long  0
0777:
0778: #
0779: # Table of Interrupt vector groups.
0780: #
0781:     .align 4
0782: IntGrps:
0783: # 0x00000800 - IL 0x00 IN 0x00; Nonmaskable interrupt.
0784:     .long  1,              I_Nmi, INT_SSW+0x00
0785: # 0x00000808 - IL 0x00 IN 0x01; Unexpected interrupt.
0786:     .long  47,             I_Unexp,      HANG_SSW+0x01
0787: # 0x00000900 - IL 0x30 IN 0x00; System clock interrupt.
0788:     .long  1,              I_Clock,      INT_SSW+0x20
0789: # 0x00000908 - IL 0x30 IN 0x01; Unexpected interrupt.
0790:     .long  207,            I_Unexp,      HANG_SSW+0x31
0791: # End of table
0792:     .long  0
```

9.4.8 Miscellaneous Support Routines

This section describes several support routines that do not fall naturally into any of the other sections.

9.4.8.1 InitGbl - Initialize Global Data

The **InitGbl** routine clears the entire global data page, copies the system name into the page, and zeroes the system clock. The Monitor does not enable interrupts until the clock is initialized.

```
0793: #
0794: # Initialize the Global Data page.
0795: #
0796:     .align 2
0797: InitGbl:
0798:     loadi  $PAGE_LEN,r0     # Page length
0799:     loada  SVAD_GDAT,r1     # Store address
0800:     loadq  $0,r2            # Initialization value
0801:     initc                   # Clear the page
0802:
0803: # Copy the system name.
0804:     loadbu _ignlen,r0       # Length of string to move
0805:     loada  _igname,r1       # Source string
0806:     loada  SVAD_GNAM,r2     # Destination string
0807:     movc                    # Copy the string
0808:
0809: # Initialize the clock.  (Do not enable interrupts before executing
0810: # this code.)
0811:     loadq  $0,r0
0812:     storw  r0,SVAD_GCLK     # Set the clock to 0
0813:     ret    sp
0814:
0815: # System name - for global data page.
0816: _igname:
0817:     .byte  NL,'E,'x,'a,'m,'p,'l,'e,' ,'S,'u,'p,'e,'r,'v,'i,'s,'o,'r
0818:     .byte  ' ,'P,'r,'o,'g,'r,'a,'m,' ,'-,' ,'V,'e,'r,'s,'i,'o,'n
0819:     .byte  ' ,'1,'.,'0,'0,NL,NL,0
0820:
0821: _ignlen:
0822:     .byte  _ignlen-_igname
```

9.4.8.2 SetPte - Set a Page Table Entry

SetPte constructs and sets one Page Table entry. It takes as parameters a real page number, a virtual page number, the System Tag and Protection Level bits (**PTE_ST** and **PTE_PL**), and the base address of the Page Table. **SetPte** validates the parameters, constructs the entry, and stores it in the requested Page Table. If the virtual page number is invalid, **SetPte** prints a diagnostic message and crashes the system.

```
0823: #
0824: # Set one Page Table Entry.
0825: # Expects:
0826: #    R0 = Real page number
0827: #    R1 = Virtual page number
0828: #    R2 = Protection Level and System Tag bits
0829: #    R3 = Base address of Page Table
0830: # Returns:
0831: #    R0 = new Page Table Entry
0832: #
0833:     .align 2
0834: SetPte:
0835:     shli   $SHF_PTE_RA,r0    # R0 = real address
0836:     cmpi   $NBR_PTE,r1       # Bounds check virtual page number
0837:     bcle   _spvpage          # Invalid
0838:     shli   $2,r1             # R1 = offset into Page Table
0839:     orw    r2,r0             # R0 = Page Table entry
0840:     storw  r0,[r1](r3)       # Set the entry
0841:     ret    sp
0842:
0843: # The virtual page number was invalid - crash the system
0844: _spvpage:
0845:     loada  _spvpmsg,r0
0846:     loadq  $PARM_LONG,r2
0847:     b      Crash
0848:
0849: _spvpmsg:
0850:     .byte  '*,' ,'S,'e,'t,'P,'t,'e,':,' ,'I,'n,'v,'a,'l,'i,'d
0851:     .byte  ' ,'P,'a,'g,'e,' ,'N,'u,'m,'b,'e,'r,' ,'=,' ,0
```

9.4.9 Memory Management Routines

This section describes the Monitor's memory allocation and deallocation routines. These routines support Main Memory, I/O Space and Boot Space.

9.4.9.1 AllocPage - Allocate One Page

AllocPage allocates one page of Main Memory, I/O Space, or Boot Space, according to the page usage value parameter (**PGU_XXXX**). A usage value of **PGU_IO** represents a request for one page of I/O Space. A usage value of **PGU_BS** represents a request for one page of Boot Space. Any other usage value represents a request for one page of Main Memory.

AllocPage calls **SetUse** to initialize the registers. **SetUse** sets **R1** to the base address of the appropriate usage table, **R2** to the base address of the appropriate use count table, and **R3** to the number of pages in the requested address space. The loop code at **_aploop** searches the appropriate usage table sequentially (from low pages to high pages), looking for an available entry (an entry with a value of **PGU_NONE**). If it finds a free page, it exits the loop to **_apfound**. Otherwise, it sets the **PSW_C** bit to indicate failure and exits through **_apdone**.

The code at **_apfound** examines the current use count for the page. If it is not zero, **AllocPage** crashes the system, since the table is corrupt. Otherwise, it sets the use count to one, stores the new page usage, clears the **PSW_C** bit, and returns.

```
0852: #
0853: # Allocate a page of Main Memory, I/O Space or Boot Space.
0854: # Expects:
0855: #    R0 = usage for page
0856: # Returns:
0857: #    PSW_C = 0, R0 = page number; success
0858: #    PSW_C = 1; failure - no pages available in requested space
0859: #
0860:      .align 2
0861: AllocPage:
0862:      call   sp,SetUse          # Set up the register for the usage info
0863:      movw   r0,r4              # R4 = new usage for page
0864:      loadq  $0,r0              # R0 = page number
0865: _aploop:
0866:      loadbu [r0](r1),r5        # R5 = current usage flag for this page
0867:      cmpq   $PGU_NONE,r5       # Is the page available?
0868:      bceq   _apfound           #  Yes - return it to the requester
0869:      addq   $1,r0              # Increment the page number
0870:      cmpw   r0,r3              # Page number < limit?
0871:      bcltu  _aploop            #  Yes - Continue search
0872:      loadi  $PSW_C,r1          # PSW_C = failure
0873:      b      _apdone
0874:
0875: _apfound:
0876:      loadbu [r0](r2),r5        # R5 = current use count for page
0877:      cmpq   $0,r5              # Is the use count 0?
0878:      bcne   _apbaduc           #  No - crash the system
0879:      loadq  $1,r5              #  Yes - set it to 1
0880:      storb  r5,[r0](r2)        #   and store it
0881:      storb  r4,[r0](r1)        # Set new page usage
0882:      loadq  $0,r1              # PSW_C = 0; success
0883: _apdone:
0884:      movwp  r1,psw
0885:      ret    sp
0886:
0887: _apbaduc:
0888:      movw   r0,r1              # R1 = page number
0889:      loada  _apbmsg,r0         # R0 = message address
0890:      loadq  $PARM_LONG,r2
0891:      b      Crash              # Crash the system
0892:
0893: _apbmsg:
0894:      .byte  '*,' ,'A,'l,'l,'o,'c,'P,'a,'g,'e,':,' ,'U,'s,'e
0895:      .byte  ' ,'C,'o,'u,'n,'t,' ,'-,' ,'P,'a,'g,'e,' ,'=,' ,0
```

9.4.9.2 FreePage - Free One Page

FreePage frees one page of Real Memory, I/O Space, or Boot Space, according to the usage implied by the Page Table entry parameter. If the **PTE_F** bit is set, the entry does not refer to a page. In that case **FreePage** returns through **_fpdone**. If the **PTE_F** bit is clear, **FreePage** sets it in the Page Table entry to indicate that the page is no longer valid. Next, **FreePage** extracts the real address (**PTE_RA**) and the System Tag bits (**PTE_ST**) from the Page Table entry, converts the address to a page number, and calls **SetUseST** to initialize the registers. **SetUseST** sets **R1** to the base address of the appropriate usage table, **R2** to the base address of the appropriate use count table, and **R3** to the number of pages in the requested address space.

The **FreePg** entry point allows callers to free pages in any of the three memory spaces, based upon a real page number, rather than a page table address. **FreePg** is called only by **FreePgRM**, which frees the pages previously reserved by calls to **ResvRM**.

FreePage examines the current usage value for the requested page. If the page is already available, the table is corrupt so **FreePage** crashes the system. If the page is in use as expected, **FreePage** acquires and decrements the page use count. If the decremented count is negative, the use count table is corrupt, so **FreePage** crashes the system. If the new count is zero, **FreePage** stores the **PGU_NONE** flag into the usage table. If the new use count is positive (indicating that the page was shared), **FreePage** does not alter the page usage value. In either case, **FreePage** stores the decremented use count and returns.

```
0896: #
0897: # Free the page associated with one Page Table Entry.  The page
0898: # can be in Main Memory, I/O Space, or Boot Space.
0899: # Expects:
0900: #    R0 = address of Page Table Entry
0901: # Returns:
0902: #    Nothing
0903:      .align 2
0904: FreePage:
0905:      loadw  (r0),r1          # R1 = Page Table Entry
0906:      movw   r1,r2            # R2 = Page Table Entry
0907:      loadi  $PTE_F,r3
0908:      andw   r3,r2            # Is the fault bit set?
0909:      brne   _fpdone          #  Yes - nothing to do
0910:      storw  r3,(r0)          #  No - set it in the Page Table Entry
0911:      movw   r1,r0            # R0 = Page Table Entry
0912:      cmpi   $PTBL_PROT,r0    # Is this a protected page?
0913:      bceq   _fpdone          #  Yes - there's no memory to free
0914:      andi   $PTE_RA,r0       # R0 = real address
0915:      shli   $-SHF_PTE_RA,r0  # R0 = real page number
0916:      andi   $PTE_ST,r1       # R1 = system tag bits
0917:      call   sp,SetUseST      # Set up the registers for the usage info
0918:
0919: # Enter here with real page number in R0 and usage registers set up.
0920: FreePg:
0921:      cmpw   r0,r3
0922:      bcgeu  _fpbadp          # page number >= limit
0923:      loadbu [r0](r1),r4      # R4 = current usage of page
0924:      loadq  $PGU_NONE,r5
0925:      cmpw   r4,r5            # Is the page already available?
0926:      bceq   _fpfree          #  Yes - crash the system
0927:      loadbu [r0](r2),r4      #  No - decrement the usage count
0928:      subq   $1,r4
0929:      brlt   _fpbaduc         # Negative use count - crash the system
0930:      brgt   _fpnofree        # Positive use count - don't free the page
0931:      storb  r5,[r0](r1)      # Make the page available
0932: _fpnofree:
0933:      storb  r4,[r0](r2)      # Set the new usage count
0934: _fpdone:
0935:      ret    sp
0936:
0937: _fpbadp:
0938:      loada  _fpbpmsg,r2
0939:      b      _fpdie
0940:
0941: _fpbaduc:
0942:      loada  _fpbucmsg,r2
0943:      b      _fpdie
```

9.4.9.3 SharePage - Shared Memory Processor

SharePage expects the virtual address of the page to share, and the relevant Page Table Directory Origin (**PDO**) address. First, **SharePage** calls the **TransVA** routine to translate the virtual address into a real address. If the address translation process results in a page fault, **SharePage** returns **PSW_C** to indicate failure. If the address was successfully translated, **SharePage** calls **SetUseST** to initialize the registers. **SetUse** sets **r1** to the base address of the appropriate usage table, **r2** to the base address of the appropriate use count table, and **r3** to the number of pages in the requested address space.

Next, **SharePage** converts the real address returned by **TransVA** into a real page number and then checks to see that the page number is valid. If not, **SharePage** crashes the system. If the page number is valid, **SharePage** retrieves the current page usage value and checks to see if the page is in use as expected. If the page is available, **SharePage** crashes the system, since the usage table is corrupt. Finally, **SharePage** retrieves the use count of the page and validates that it is non-zero. If not, **SharePage** crashes the system because, again, the table is corrupt. If the count is as expected, **SharePage** increments it, stores it back into the use count table, and returns.

```
0944: _fpfree:
0945:     loada  _fpfmsg,r2
0946: _fpdie:
0947:     movw   r0,r1              # R1 = page number
0948:     movw   r2,r0              # R0 = message text
0949:     loadq  $PARM_LONG,r2
0950:     b      Crash              # Crash the system
0951:
0952: _fpbpmsg:
0953:     .byte  'F,'r,'e,'e,'P,'a,'g,'e,':,' ,'I,'n,'v,'a,'l,'i,'d
0954:     .byte  ' ,'P,'a,'g,'e,' ,'=,' ,0
0955:
0956: _fpbucmsg:
0957:     .byte  'F,'r,'e,'e,'P,'a,'g,'e,':,' ,'U,'s,'e,' ,'C,'o,'u,'n,'t
0958:     .byte  ' ,'-,' ,'P,'a,'g,'e,' ,'=,' ,0
0959:
0960: _fpfmsg:
0961:     .byte  'F,'r,'e,'e,'P,'a,'g,'e,':,' ,'A,'l,'r,'e,'a,'d,'y
0962:     .byte  ' ,'F,'r,'e,'e,' ,'-,' ,'P,'a,'g,'e,' ,'=,' ,0
0963:
0964: #
0965: # Share a page.
0966: # Expects:
0967: #   R0 = virtual address of page to share
0968: #   R1 = relevant Page Table Directory Origin address
0969: # Returns:
0970: #   PSW_C = 0, R0 = real page number of shared page
0971: #   PSW_C = 1, failure
0972: #
0973:     .align 2
0974: SharePage:
0975:     call   sp,TransVA         # R0 = PTE_RA, R1 = PTE_ST
0976:     bc     _spdone            # Failed
0977:     call   sp,SetUseST        # Set the usage table and use count table addrs
0978:     shli   $-SHF_PTE_RA,r0    # R0 = real page number
0979:     cmpw   r0,r3
0980:     bcgeu  _spbadp            # page number >= limit
0981:     loadbu [r0](r1),r4        # R4 = current usage of page
0982:     cmpq   $PGU_NONE,r4
0983:     bceq   _spfree            # Page is free - crash the system
0984:     loadbu [r0](r2),r4        # R4 = current use count
0985:     cmpq   $0,r4              # Is it 0?
0986:     breq   _spbaduc           #  Yes - crash the system
0987:     addq   $1,r4              #  No - increment it
0988:     storb  r4,[r0](r2)        #   and store it
0989: _spdone:
0990:     ret    sp
```

The code on the facing page processes any error conditions detected by **SharePage**. The routines set up the registers for the **Crash** routine and then branch to it unconditionally.

```
0991: _spbadp:
0992:     loada  _spbpmsg,r2
0993:     b      _spdie
0994:
0995: _spbaduc:
0996:     loada  _spbucmsg,r2
0997:     b      _spdie
0998:
0999: _spfree:
1000:     loada  _spfmsg,r2
1001: _spdie:
1002:     movw   r0,r1              # R1 = page number
1003:     movw   r2,r0              # R0 = message text
1004:     loadq  $PARM_LONG,r2
1005:     b      Crash              # Crash the system
1006:
1007: _spbpmsg:
1008:     .byte  'S,'h,'a,'r,'e,'P,'a,'g,'e,':,' ,'I,'n,'v,'a,'l,'i,'d
1009:     .byte  ' ,'P,'a,'g,'e,' ,'=,' ,0
1010:
1011: _spbucmsg:
1012:     .byte  'S,'h,'a,'r,'e,'P,'a,'g,'e,':,' ,'U,'s,'e,' ,'C,'o,'u,'n,'t
1013:     .byte  ' ,'-,' ,'P,'a,'g,'e,' ,'=,' ,0
1014:
1015: _spfmsg:
1016:     .byte  'S,'h,'a,'r,'e,'P,'a,'g,'e,':,' ,'A,'l,'r,'e,'a,'d,'y
1017:     .byte  ' ,'F,'r,'e,'e,' ,'-,' ,'P,'a,'g,'e,' ,'=,' ,0
```

9.4.9.4 TransVA - Translate Virtual Address

TransVA expects a virtual address in **r0**, and the Page Table Directory Origin (**PDO**) address in **r1**. The routine translates virtual address into real addresses using the same algorithm that the CAMMU employs. First, **TransVA** extracts the Page Table Directory Index (**VA_PTDI**) field from the virtual address and converts it to an indexed offset into the Page Table Directory. The code line 1030 performs the conversion.

The operation is clearer if you think of it in two steps:

1. Shift the Page Table Directory Index field right by **SHF_VA_PTDI** to convert it into an index.
2. Shift the index left by two bits (that is, multiply it by four) to convert it into a Page Table Directory offset.

TransVA then retrieves the Page Table Directory entry, makes a copy of it in register **r3**, and checks the **PTDE_F** bit. If it is set, there is no valid Page Table for the requested virtual address, so **TransVA** sets the **PSW_C** bit to indicate a translation failure and returns. If the Page Table Directory entry references a valid Page Table, **TransVA** isolates the Page Table Origin address (**PTDE_PTO**) in register **r2**.

Next, **TransVA** isolates the Page Table Index field from the virtual address and converts it into a Page Table offset, exactly as it did for the Page Table Directory offset above. Then **TransVA** retrieves the Page Table entry and checks the **PTE_F** bit. If it is set, there is no valid page allocated to the requested virtual address, so **TransVA** sets the **PSW_C** bit to indicate a translation failure and returns. If there is a page at this virtual address, **TransVA** isolates both the System Tag (**PTE_ST**), and real address (**PTE_RA**) fields, clears the **PSW_C** bit to indication a successful translation, and returns.

The two pages following the code contain diagrams and narrative that clarify the CLIPPER virtual to physical address translation algorithm.

```
1018: # Translate virtual address to real address.
1019: # Expects:
1020: #    R0 = Virtual address
1021: #    R1 = Page Table Directory origin
1022: # Returns:
1023: #    PSW_C = 0, R0 = real address, R1 = System Tag
1024: #    PSW_C = 1, page fault
1025: #
1026:     .align 2
1027: TransVA:
1028:     movw   r0,r2            # R2 = copy of virtual address
1029:     andi   $VA_PTDI,r2      # R2 = Page Table Directory index
1030:     shli   $(-SHF_VA_PTDI+2),r2 # R2 = Page Table Directory offset
1031:     loadw  [r2](r1),r2      # R2 = Page Table Directory entry
1032:     movw   r2,r3
1033:     andi   $PTDE_F,r3       # Page fault?
1034:     brne   _tvfail          #  Yes
1035:     andi   $PTDE_PTO,r2     # R2 = Page Table address
1036:     andi   $VA_PTI,r0       # R0 = Page Table index
1037:     shli   $(-SHF_VA_PTI+2),r0  # R0 = Page Table offset
1038:     loadw  [r0](r2),r0      # R0 = Page Table entry
1039:     movw   r0,r1
1040:     andi   $PTE_F,r1        # Page fault?
1041:     brne   _tvfail          #  Yes
1042:     movw   r0,r1            # R1 = Page Table entry
1043:     andi   $PTE_ST,r1       # R1 = System Tag
1044:     andi   $PTE_RA,r0       # R0 = real address
1045:     loadq  $0,r2            # PSW_C = 0, success
1046: _tvdone:
1047:     movwp  r2,psw
1048:     ret    sp
1049:
1050: _tvfail:
1051:     loadi  $PSW_C,r2        # PSW_C = 1, failure
1052:     b      _tvdone
```

Figure 9.9 illustrates the CLIPPER's virtual to physical address translation algorithm. The hardware within the CAMMU performs the translation as follows:

1. The Dynamic Translation Unit uses the appropriate CAMMU Page Table Directory Origin register (**C_SPDO** or **C_UPDO**) to find the base address of the Page Table Directory in real memory.
2. The DTU extracts the Page Table Directory Index (**VA_PTDI**) from the virtual address, multiplies it by four, and adds the result to the Page Table Directory Origin obtained in step 1. The resulting value is the real memory address of the Page Table Directory entry.
3. The DTU fetches the Page Table Directory entry from real memory. If the fault bit (**PTDE_F**) is set, the DTU generates a data page fault. Otherwise, the DTU extracts the Page Table Origin (**PTDE_PTO**) from the Page Table Directory entry and uses it as the base address of the Page Table in real memory.
4. The DTU extracts the Page Table Index field (**VA_PTI**) from the virtual address, multiplies it by four, and adds the result to the Page Table Origin obtained in step 3. The resulting value is the real memory address of the Page Table entry.
5. The DTU fetches the Page Table entry from real memory. If the fault bit (**PTE_F**) is set, the DTU generates a page fault exception. Otherwise, the DTU extracts the Real Address (**PTE_RA**) from the Page Table entry and uses it as the base address of the target page in real memory.
6. The DTU extracts the Page Offset field (**VA_OFFS**) from the virtual address and adds it to the base address of the target page obtained in step 5. The resulting value is the translated real memory address.

Figure 9.9 Virtual to Real Address Translation

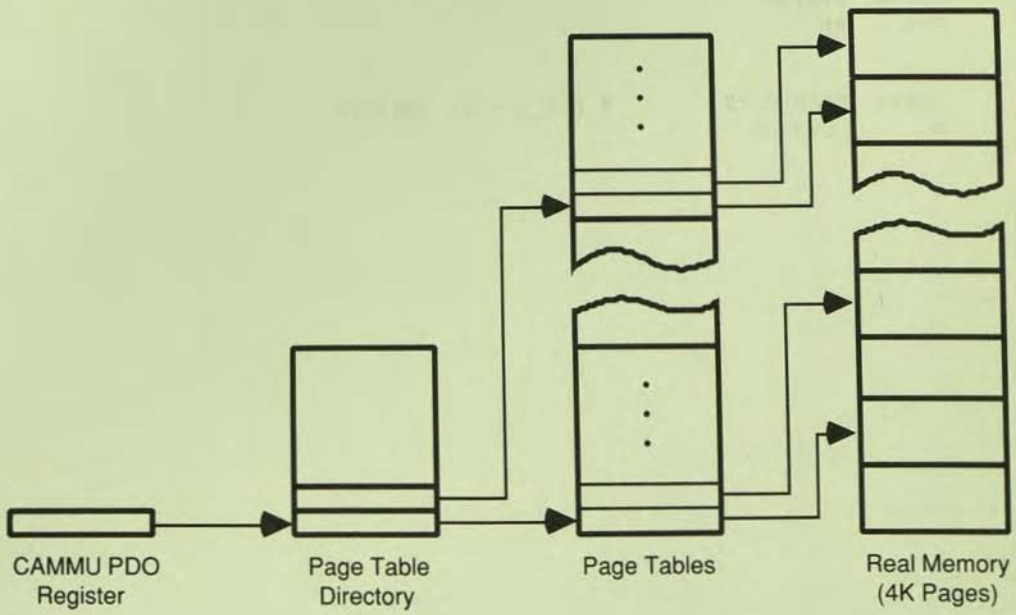
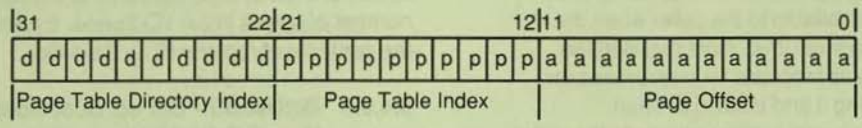
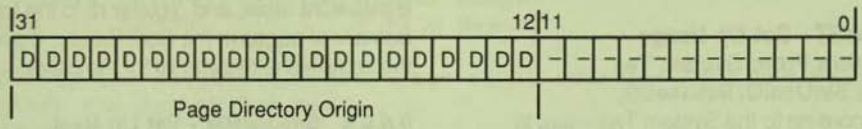


Figure 9.10 illustrates the fields within each of the translation registers. It also shows the arithmetic performed in the translation process.

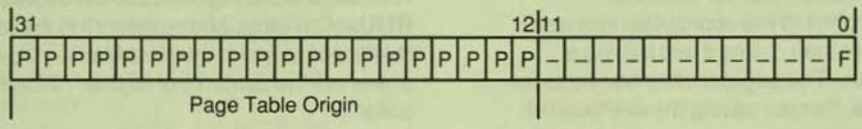
Figure 9.10 Virtual to Real Address Calculation



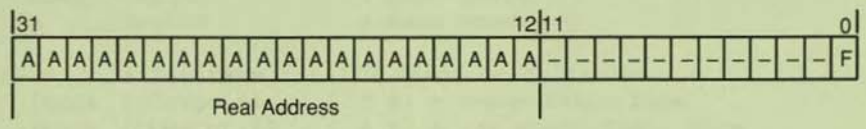
Virtual Address



CAMMU Page Directory Origin Register



Page Table Directory Entry



Page Table Entry

```
DDDDDDDDDDDDDDDDDDD   000000000000 – Page Directory Origin
+00000000000000000000 dddddddddd00 – Page Tbl Dir Index
----------------------------------
DDDDDDDDDDDDDDDDDDD   dddddddddd00 – Page Tbl Dir Entry Addr

PPPPPPPPPPPPPPPPPPP   000000000000 – Page Table Origin
+00000000000000000000 pppppppppp00 – Page Table Index
----------------------------------
PPPPPPPPPPPPPPPPPPP   pppppppppp00 – Page Table Entry Addr

AAAAAAAAAAAAAAAAAAA   000000000000 – Page Address
+00000000000000000000 aaaaaaaaaaaa – Page Offset
----------------------------------
AAAAAAAAAAAAAAAAAAA   aaaaaaaaaaaa – Real Memory Address
```

9.4.9.5 SetUse - Set Up Usage Registers

SetUse calls **SetUseIO**, **SetUseBS**, **SetUseRM**, according to the page usage value in register **r0**. It also preserves the contents of **r0**, so that it is still available to the caller when the request processor returns. Note that **SetUse** branches to the appropriate request processor, rather than calling it and issuing a return instruction. The target routine returns to the caller of **SetUse**, thereby saving the overhead of one call/return sequence.

9.4.9.6 SetUseST - Set Up Usage Registers From System Tag

SetUseST calls **SetUseIO**, **SetUseBS**, **SetUseRM**, according to the System Tag value in register **r1**. It preserves the contents of **r0** (but not **r1**), so that it is still available to the caller when the request processor returns. Note that **SetUseST** branches to the appropriate request processor, rather than calling it and issuing a return instruction. The target routine returns to the caller of **SetUse**, thereby saving the overhead of one call/return sequence.

9.4.9.7 SetUseIO - Set Up I/O Space Usage Registers

SetUseIO sets register **r1** to the address of the **IOUsage** table, register **r2** to the address of the **IOUseCnt** table, and register **r3** to the total number of pages in the I/O Space. It preserves the contents of register **r0** for the caller.

9.4.9.8 SetUseBS - Set Up Boot Space Usage Registers

SetUseBS sets register **r1** to the address of the **BSUsage** table, register **r2** to the address of the **BSUseCnt** table, and register **r3** to the total number of pages in the Boot Space. It preserves the contents of register **r0** for the caller.

9.4.9.9 SetUseRM - Set Up Real Memory Usage Registers

SetUseRM sets register **r1** to the address of the **RMUsage** table, register **r2** to the address of the **RMUseCnt** table, and register **r3** to the total number of pages in the Real Memory Space. It preserves the contents of register **r0** for the caller.

```
1053: # Set up the usage table and usage count addresses and the page bounds
1054: # given the page usage value.
1055: # Expects:
1056: #    R0 = page usage value
1057: # Returns:
1058: #    R0 = preserved
1059: #    R1 = usage table address
1060: #    R2 = use count table address
1061: #    R3 = page limit value
1062: SetUse:
1063:      cmpq   $PGU_IO,r0
1064:      bceq   SetUIO            # I/O Space
1065:      cmpq   $PGU_BS,r0
1066:      bceq   SetUBS            # Boot Space
1067:      b      SetURM            # Main Memory
1068:
1069: # Set up the usage table and usage count addresses and the page bounds
1070: # given the System Tag bits.
1071: # Expects:
1072: #    R1 = PTE_ST
1073: # Returns:
1074: #    R0 = preserved
1075: #    R1 = usage table address
1076: #    R2 = use count table address
1077: #    R3 = page limit value
1078: SetUseST:
1079:      cmpi   $PTE_ST_NCIO,r1
1080:      bceq   SetUIO            # I/O Space
1081:      cmpi   $PTE_ST_NCBS,r1
1082:      bceq   SetUBS            # Boot Space
1083:      b      SetURM            # Main Memory
1084:
1085: SetUIO:    # Set regs for I/O Space usage tables.
1086:      loada  IOUsage,r1        # R1 = usage table base
1087:      loada  IOUseCnt,r2       # R2 = use count table base
1088:      loadi  $IO_PAGES,r3      # R3 = limit
1089:      ret    sp
1090:
1091: SetUBS:    # Set regs for Boot Space usage tables.
1092:      loada  BSUsage,r1        # R1 = usage table base
1093:      loada  BSUseCnt,r2       # R2 = use count table base
1094:      loadi  $BS_PAGES,r3      # R3 = limit
1095:      ret    sp
1096:
1097: SetURM:    # Set regs for Main Memory usage tables.
1098:      loada  RMUsage,r1        # R1 = usage table base
1099:      loada  RMUseCnt,r2       # R2 = use count table base
1100:      loadi  $RM_PAGES,r3      # R3 = limit
1101:      ret    sp
```

9.4.9.10 ResvRM - Reserve Real Memory Page

The Monitor calls **ResvRM** to reserve certain real memory pages so that they are unavailable for allocation. For the requested page, **ResvRM** simply sets the **RMUsage** table entry to **PGU_RESV**, and the **RMUseCnt** value to one. The routine is meant to be called once, before memory allocation begins, so it performs no error checking.

Before memory allocation begins, the Monitor reserves real memory pages 4, 5, 6, and 7 so that they are unavailable for allocation. These four pages are "hidden" behind the HTLB I/O and Boot Space pages. The Monitor cannot access real memory pages 4 through 7 in unmapped mode, since pages 0 through 7 are always translated through the HTLB. Since the Monitor uses the normal memory allocation routines to set up its own page table, pages 4 through 7 must be removed until the Monitor's virtual memory is established. Once the Page Tables are intiialized, the Monitor calls **FreeRM** to release the memory for normal allocation.

9.4.9.11 FreeRM - Free Real Memory Page.

The Monitor calls **FreeRM** to release the reserved real memory pages after the Page Tables are initialized. **FreeRM** calls **SetUseRM** to set up the usage registers and then branches to **FreePg**, which is within the **FreePage** routine. **FreePg** returns to the caller of **FreeRM**.

```
1102: #
1103: # Reserve a page of Real Memory.
1104: # Expects:
1105: #    R0 = Page number to reserve
1106: # Returns:
1107: #    Nothing
1108: #
1109:      .align 2
1110: ResvRM:
1111:      loadq  $PGU_RESV,r1
1112:      storb  r1,RMUsage(r0)    # Reserve the page
1113:      loadq  $1,r1
1114:      storb  r1,RMUseCnt(r0)
1115:      ret    sp
1116:
1117: #
1118: # Free a page of Real Memory.
1119: # Expects:
1120: #    R0 = Page number to free
1121: # Returns:
1122: #    Nothing
1123: #
1124:      .align 2
1125: FreeRM:
1126:      call   sp,SetURM         # Set up the usage registers
1127:      b      FreePg            # Free it
```

9.4.10 User Virtual Memory Support

Before it can execute a user program, the Monitor must establish a virtual memory environment for it, and read its code and data from the **BinIn** device. The code in this section performs all of these tasks.

9.4.10.1 The User Program Environment

Figure 9.11 illustrates the virtual memory environment that the Monitor creates for each user program.

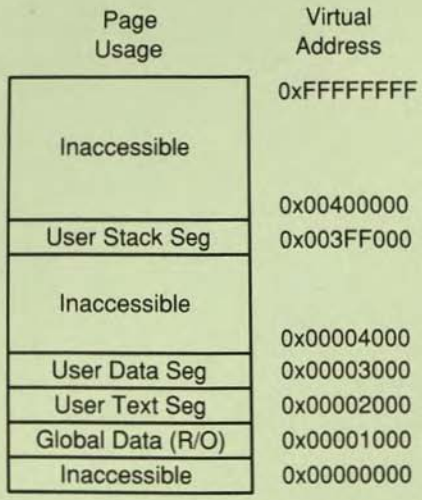
Note that virtual page 0 is inaccessible. A frequent error made by high level programmers is to dereference an uninitialized pointer. This usually results in fetching or storing data indirectly through location 0. If this happens to result in accessing a legal (but incorrect) address, the program is likely to continue for thousands of instructions past the point of the initial error. Obviously, errors of this nature are extremely difficult to find and fix. In this virtual memory environment, all references to location 0 generate a protection fault (that is, a read, write, or execution protection trap, depending upon the type of access attempted). The address of the offending instruction is printed by the trap handler, so the programmer can locate the problem.

Virtual page 1 is set up to point to the Monitor's global data page. This page holds information that is maintained by the Monitor, but may be of interest to the user program. This is a shared page: the Monitor has read/write access to it, but the user is restricted to reading to data only.

Virtual page 2 always contains the user program's code (text). The Monitor assumes that every user program contains only one page of code.

Virtual page 3 contains the user program's data. Again, the Monitor assumes that every user program starts with only one page of data. The user program can extend its data segment with the **SYS_EXDS** system call.

Figure 9.11 Initial User Virtual Memory Usage



Virtual memory between pages 4 and 1,022 (0x3FE) is inaccessible and generates page faults when referenced. These virtual addresses are available for the user's data and stack segments, should they need to grow. The Monitor provides a system call (**SYS_EXDS**) that allows the user to extend his data segment one page at a time. There is no call available to return user memory to the free space pool.

Finally, virtual page 1,023 (0x3FF) contains the stack. The Monitor initally allocates one page for the user's stack. It allocates additional pages as the program attempts to reference them. The data page fault trap handler checks to see whether a stack operation caused the fault. If so, it automatically extends the stack (downward - the CLIPPER stack grows from high to low addresses) to include the address that generated the fault. The stack is isolated in high virtual memory so that stack underflow and overflow conditions can be diagnosed by the hardware through the page fault mechanism.

Although the Monitor takes advantage of the CLIPPER's memory mapping capabilities, it does not implement demand-paged virtual memory. Thus, all of the user's code and data remain resident in memory from the time they are loaded from the **BinIn** device until the program exits.

9.4.10.2 LoadUPgm - Load a User Program

LoadUPgm simply loads the user's protected, global, text (code), data, and stack segments and returns. Loading implies different things for each segment. The documentation for the individual load routines explains the differences.

9.4.10.3 LoadUGbl - Load the User's Global Data Segment

LoadUGbl is responsible for creating a shared memory page at user virtual page **UVPN_GDAT** that references that system global data page. To accomplish this task, **LoadUGbl** calls **SharePage** with the virtual address of the global data page in Monitor space. **SharePage** translates the Monitor virtual address into a real page number and increments the **RMUseCnt** field to indicate a shared page. **SharePage** returns a failure indication (**PSW_C** bit set) if it could not translate the virtual address. Since this address is an assembly time constant, it must be valid. **LoadUGbl** crashes the system if **SharePage** fails.

Next, **LoadUGbl** calls **SetPte** to set the user's Page Table entry, thereby installing the global page into the user's virtual memory map. Finally, **LoadUGbl** clears the **PSW_C** bit to indicate success and returns.

```
1128: #
1129: # Load a user program.
1130: # Expects:
1131: #   Nothing
1132: # Returns:
1133: #   PSW_C = 0; success
1134: #   PSW_C = 1; failure - no memory available
1135: #
1136:     .align 2
1137: LoadUPgm:
1138:     call   sp,LoadUProt      # Load the protected page
1139:     call   sp,LoadUGbl       # Load the global data segment
1140:     call   sp,LoadUText      # Load the user's text segment
1141:     bc     _lupdone          # Failed to load the text segment
1142:     call   sp,LoadUData      # Load the user's data segment
1143:     bc     _lupdone          # Failed to load the data segment
1144:     call   sp,LoadUStk       # Load the user's stack segment
1145: _lupdone:
1146:     ret    sp                # PSW_C set to indicate success or failure
1147:
1148: #
1149: # Load the global segment of a user program.  Assumes that the global
1150: # segment contains exactly one page.
1151: # Expects:
1152: #   Nothing
1153: # Returns:
1154: #   PSW_C = 0; success
1155: #   PSW_C = 1; failure
1156: #
1157:     .align 2
1158: LoadUGbl:
1159:     loada  SVAD_GDAT,r0      # R0 = Supv virtual address of global data
1160:     loada  SVAD_SPTD,r1      # R1 = base address of Supv Page Table Dir
1161:     call   sp,SharePage      # R0 = real page number of shared page
1162:     bc     _lugfault         # Page fault on virtual address translation
1163:     loadi  $UVPN_GDAT,r1     # User virtual page number of global data
1164:     loadi  $PTBL_UGBL,r2     # R2 = system tag and protection
1165:     loada  SVAD_UPT,r3       # R3 = base address of Page Table
1166:     call   sp,SetPte         # Set the user's Page Table Entry
1167:     loadq  $0,r0             # PSW_C = 0
1168:     movwp  r0,psw
1169:     ret    sp
1170:
1171: _lugfault:
1172:     loada  _lugmsg,r0
1173:     loadq  $PARM_NULL,r2
1174:     b      Crash             # Crash the system
1175:
1176: _lugmsg:
1177:     .byte  '*,' ,'L,'o,'a,'d,'U,'G,'l,'b,'l,':,' ,'G,'l,'o,'b,'a,'l
1178:     .byte  ' ,'D,'a,'t,'a,' ,'P,'a,'g,'e,' ,'F,'a,'u,'l,'t,NL,0
```

9.4.10.4 LoadUText - Load the User's Text Segment

LoadUText is responsible for reading one page of data from the **BinIn** device and installing the allocated page into the user's virtual memory space. To accomplish this task, **LoadUText** calls **LoadPage** with the **PGU_TEXT** page usage value. **LoadPage** allocates a page of real memory and fills it with data from the **BinIn** device. Then, **LoadUText** calls **SetPte** to install the new page into the user's virtual memory map. Finally, **LoadUText** clears the **PSW_C** bit to indicate success and returns.

9.4.10.5 LoadUData - Load the User's Data Segment

LoadUData is responsible for reading one page of data from the **BinIn** device and installing the allocated page into the user's virtual memory space. To accomplish this task, it calls **LoadPage** with the **PGU_DATA** page usage value. **LoadPage** allocates a page of real memory and fills it with data from the **BinIn** device. Next, **LoadUData** sets the **U_DataLen** variable to 1, and the **U_TDataPg** (the top page of user data) to **UVPN_DATA**. The Monitor maintains this information because the user program is allowed to extend its data space with the **SYS_EXDS** system call.

LoadUData then calls **SetPte** to install the new page into the user's virtual memory map. Finally, **LoadUData** clears the **PSW_C** bit to indicate success and returns.

```
1179: #
1180: # Load the text segment of a user program.  Assumes that the text
1181: # segment of every user program contains exactly one page.
1182: # Expects:
1183: #    Nothing
1184: # Returns:
1185: #    PSW_C = 0; success
1186: #    PSW_C = 1; failure - no memory available
1187: #
1188:      .align 2
1189: LoadUText:
1190:      loadq  $PGU_TEXT,r0      # R0 = page usage
1191:      call   sp,LoadPage       # Load a page from SysIn into the window
1192:      bc     _lutdone          # Failure
1193:      loadi  $UVPN_TEXT,r1     # Virtual page number of text segment
1194:      loadi  $PTBL_TEXT,r2     # R2 = system tag and protection
1195:      loada  SVAD_UPT,r3       # R3 = base address of Page Table
1196:      call   sp,SetPte         # Set the user's Page Table Entry
1197:      loadq  $0,r0
1198:      movwp  r0,psw            # PSW_C = 0
1199: _lutdone:
1200:      ret    sp
1201:
1202: #
1203: # Load the data segment of a user program.  Assumes that the data
1204: # segment of every user program contains exactly one page.
1205: # Expects:
1206: #    Nothing
1207: # Returns:
1208: #    PSW_C = 0; success
1209: #    PSW_C = 1; failure - no memory available
1210: #
1211:      .align 2
1212: LoadUData:
1213:      loadq  $PGU_DATA,r0      # R0 = page usage
1214:      call   sp,LoadPage       # Load a page from SysIn into the window
1215:      bc     _luddone          # Failure
1216:      loadq  $1,r1
1217:      storh  r1,U_DataLen      # Set the length of the data segment
1218:      loadi  $UVPN_DATA,r1     # Virtual page number of data segment
1219:      storh  r1,U_TDataPg      # Set the top page of the data segment
1220:      loadi  $PTBL_DATA,r2     # R2 = system tag and protection
1221:      loada  SVAD_UPT,r3       # R3 = base address of Page Table
1222:      call   sp,SetPte         # Set the user's Page Table Entry
1223:      loadq  $0,r0
1224:      movwp  r0,psw            # PSW_C = 0
1225: _luddone:
1226:      ret    sp
```

9.4.10.6 LoadUStk - Load the User's Stack Segment

LoadUStk is responsible for allocating one page of real memory and installing into the user's virtual memory space. **LoadUStk** calls **AllocPage** with the **PGU_STK** page usage value. **LoadUStk** returns with the **PSW_C** bit set if **AllocPage** fails. If the page was allocated, **LoadUStk** sets the stack length (**U_StkLen**) and the bottom page of the stack (**U_BStkPg**). Then it calls **SetPte** to alter the user's virtual memory map, clears the **PSW_C** bit to indicate success, and returns.

9.4.10.7 LoadUProt - Load the User's Protected Page

LoadUProt is responsible for initializing user virtual page zero so that it allows no accesses. It sets up the registers as required and calls **SetPte** to install the page in the user's Page Table.

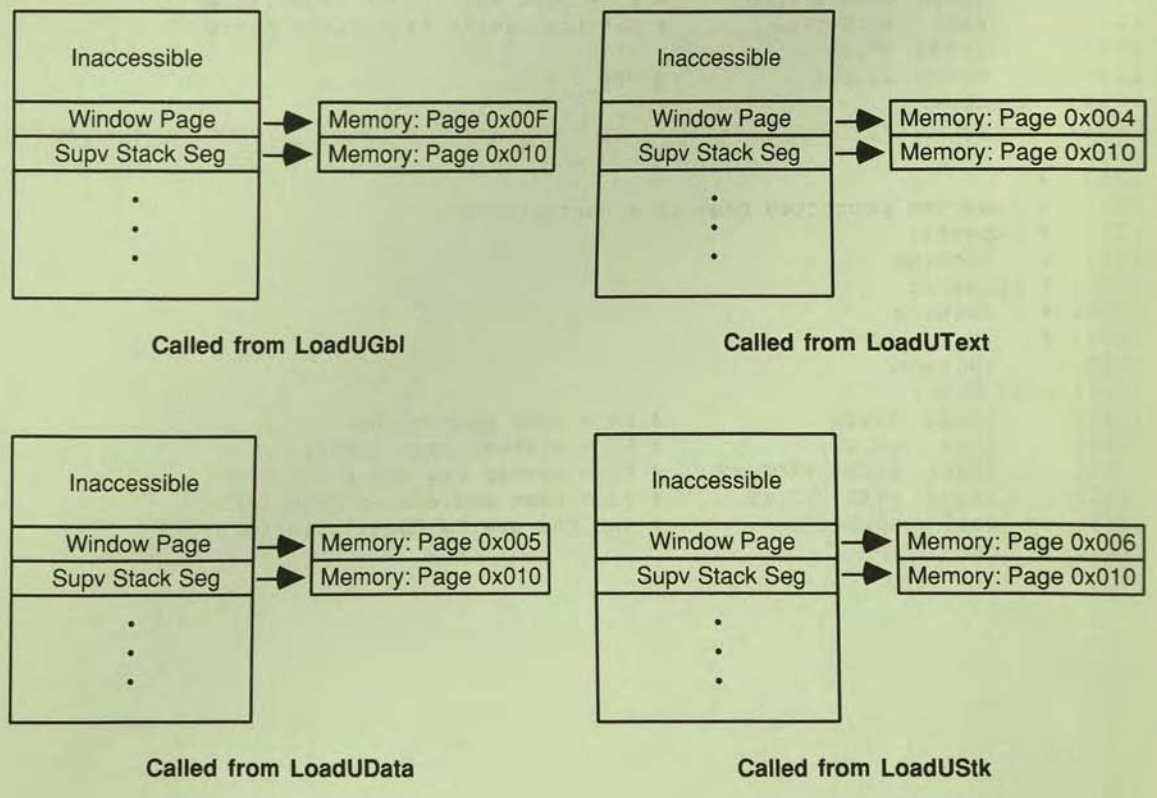
```
1227: #
1228: # Load the stack segment of a user program.  Assigns one page to
1229: # the user's stack initially.
1230: # Expects:
1231: #   Nothing
1232: # Returns:
1233: #   PSW_C = 0; success
1234: #   PSW_C = 1; failure - no memory available
1235: #
1236:     .align 2
1237: LoadUStk:
1238:     loadq  $PGU_STK,r0      # R0 = page usage
1239:     call   sp,AllocPage     # Try to get a page
1240:     bc     _lusdone         # No memory available
1241:     loadq  $1,r1
1242:     storh  r1,U_StkLen      # Set the length of the stack segment
1243:     loadi  $UVPN_STK,r1     # Virtual page number of stack segment
1244:     storh  r1,U_BStkPg      # Set the bottom page of the stack segment
1245:     loadi  $PTBL_STK,r2     # R2 = system tag and protection
1246:     loada  SVAD_UPT,r3      # R3 = base address of Page Table
1247:     call   sp,SetPte        # Set the user's Page Table Entry
1248:     loadq  $0,r0
1249:     movwp  r0,psw           # PSW_C = 0
1250: _lusdone:
1251:     ret    sp
1252:
1253: #
1254: # Load the protected page of a user program.
1255: # Expects:
1256: #   Nothing
1257: # Returns:
1258: #   Nothing
1259: #
1260:     .align 2
1261: LoadUProt:
1262:     loadq  $0,r0            # R0 = real page number
1263:     movw   r0,r1            # R1 = virtual page number
1264:     loadi  $PTBL_PROT,r2    # R2 = system tag and protection
1265:     loada  SVAD_UPT,r3      # R3 = base address of Page Table
1266:     call   sp,SetPte        # Set the user's Page Table Entry
1267:     ret    sp
```

9.4.10.8 LoadPage - Load One Page From BinIn

LoadPage is responsible for allocating one page of real memory and filling it with data from the **BinIn** device. **LoadPage** expects the new page usage value in register **r0**. It calls **AllocPage** to allocate one page, and then calls **SetWindow** to install the new page into the Monitor's virtual address space. The window is a dedicated page of virtual memory used to access different pages of real memory. Figure 9.12 illustrates the window's mapping across repeated calls to **LoadPage**:

After setting up the window, **LoadPage** calls **ReadPage** to transfer the data from the **BinIn** device. After the data has been read in, **LoadPage** calls **ClrWindow** to set the **PTE_F** bit in the window's Page Table entry. The Monitor never leaves unused pages mapped into its virtual memory space. Because of a bug or an invalid pointer, the Monitor could alter a user page inadvertently through the window. It would be extremely difficult to discover why a user program suddenly aborted, or worse, generated invalid results, after working correctly for months.

Figure 9.12 Window Map During LoadPage Call



```
1268: #
1269: # Load a page from the BinIn device.
1270: # Expects:
1271: #   R0 = new page usage
1272: # Returns:
1273: #   PSW_C = 0, R0 = real page number; success
1274: #   PSW_C = 1; failure - no memory available
1275: #
1276:     .align 2
1277: LoadPage:
1278:     call   sp,AllocPage      # Allocate a real memory page
1279:     bc     _lpdone           # No memory available
1280:     pushw  r0,sp             # Save the real page number
1281:     loadi  $PTBL_INIT,r2     # Initialization page
1282:     call   sp,SetWindow      # Set the window page to the real page
1283:     loada  SVAD_WNDO,r0      # Destination address of read
1284:     call   sp,ReadPage       # Read a page from the SysIn device
1285:     call   sp,ClrWindow      # Clear the window page
1286:     loadq  $0,r0
1287:     movwp  r0,psw            # PSW_C = 0
1288:     popw   sp,r0             # Real page number
1289: _lpdone:
1290:     ret    sp
```

9.4.10.8. ClrWindow - Clear the Window Page

ClrWindow loads the **PTE_F** value in register **r0** and branches unconditionally to **SetWindow** to write the window Page Table entry. This results in a page fault whenever the window is referenced. As explained in the previous section, the Monitor only uses the window when absolutely necessary. The Monitor calls **ClrWindow** when it no longer needs access to the page.

9.4.10.9. SetWindow - Setup the Window Page

SetWindow sets the window page to point to a given page of real memory. It expects a real page number in register **r0**, and the System Tag and Protection Level bits in register **r2**. **SetWindow** calls **SetPte** to install the new page (temporarily) into the Monitor's virtual memory map.

9.4.10.10. FreePRPT - Free Pages Referenced by Page Table

FreePRPT frees all of the I/O, Boot, and Main Memory space pages referenced by one page table. It does not free the Page Table itself, only the pages that the Page Table references. **FreePRPT** expects the base address of a Page Table in register **r0**. It simply loops through the entire table, calling **FreePage** for each entry.

```
1291: #
1292: # Clear the window page so that it generates a page fault.
1293: # Expects:
1294: #    Nothing
1295: # Returns:
1296: #    Nothing
1297: #
1298:      .align 2
1299: ClrWindow:
1300:      loadq  $0,r0              # No real page number
1301:      loadi  $PTE_F,r2          # Page fault
1302:      b      SetWindow
1303:
1304: #
1305: # Set the window page to point to a real page.
1306: # Expects:
1307: #    R0 = real page number
1308: #    R2 = protection level and system tag
1309: # Returns:
1310: #    Nothing
1311: #
1312:      .align 2
1313: SetWindow:
1314:      loadi  $SVPN_WNDO,r1      # R1 = virtual page number of window
1315:      loada  SVAD_SPT,r3        # R3 = base address of Page Table
1316:      call   sp,SetPte          # Set the Page Table Entry
1317:      ret    sp
1318:
1319: #
1320: # Free all of the pages referenced by one page table
1321: # Expects:
1322: #    R0 = Page Table address
1323: # Returns:
1324: #    Nothing
1325: #
1326:      .align 2
1327: FreePRPT:
1328:      loadi  $NBR_PTE,r1        # R1 = count
1329: _fptloop:
1330:      pushw  r1,sp
1331:      pushw  r0,sp
1332:      call   sp,FreePage        # Free the memory associated with the PTE
1333:      popw   sp,r0
1334:      popw   sp,r1
1335:      addq   $4,r0              # Increment the Page Table address
1336:      subq   $1,r1              # Decrement the count
1337:      brne   _fptloop           # Process the next one
1338:      ret    sp                 # All done
```

9.4.10.11. InitUReg - Initialize User's Registers

InitUReg loads initial values into each of the CLIPPER user registers, and also, into the floating-point registers. The Monitor program does not use the floating-point registers, so it does not need to preserve them across context switches. If the Monitor program required the floating registers, it would preserve them if and only if the **SSW_FRD** (floating registers dirty) bit was set in the **SSW**.

InitUReg loads the address of the table of initial global registers (**_gregtbl**) into **r0** and executes a **restur r0** instruction. Then, it saves the current stack point, loads the address of the table of initial floating registers (**_fregtbl**) into **SP** and executes a **restd0** instruction. Finally, **InitUReg** restores the stack pointer and returns. Note that the initial **PC** value for the user program is fixed. This means that the Monitor does not support alternate entry points for user program. They must be linked to begin execution at user virtual address **UVAD_TEXT**, which is fixed at 0x0002000.

```
1339: #
1340: # Initialize the user registers.
1341: # Expects:
1342: #   Nothing
1343: # Returns:
1344: #   Nothing
1345: #
1346:     .align 2
1347: InitUReg:
1348:     loada  _gregtbl,r0      # Table of initial general register values
1349:     restur r0
1350:     movw   sp,r0
1351:     loada  _fregtbl,sp      # Table of initial floating point reg values
1352:     restd0
1353:     movw   r0,sp
1354:     ret    sp
1355:
1356:     .align 4
1357: _gregtbl:
1358: UserR0:
1359:     .long  INIT_U_REG,INIT_U_REG,INIT_U_REG,INIT_U_REG# R0 - R3
1360:     .long  INIT_U_REG,INIT_U_REG,INIT_U_REG,INIT_U_REG# R4 - R7
1361:     .long  INIT_U_REG,INIT_U_REG,INIT_U_REG,INIT_U_REG# R8 - R11
1362:     .long  INIT_U_REG,INIT_U_REG,INIT_U_REG,INIT_U_SP # R12 - R15
1363:
1364:     .align 8
1365: _fregtbl:
1366:     .long  INIT_U_FPL,INIT_U_FPH,INIT_U_FPL,INIT_U_FPH# F0 - F1
1367:     .long  INIT_U_FPL,INIT_U_FPH,INIT_U_FPL,INIT_U_FPH# F2 - F3
1368:     .long  INIT_U_FPL,INIT_U_FPH,INIT_U_FPL,INIT_U_FPH# F4 - F5
1369:     .long  INIT_U_FPL,INIT_U_FPH,INIT_U_FPL,INIT_U_FPH# F6 - F7
```

9.4.10.12. The Memory Environment After Loading the User

Figure 9.13 illustrates the user's virtual memory map before the program begins execution:

If you compare this diagram with Figure 9.5, which illustrates real memory usage immediately after Monitor initialization, you will see that memory page 0x00F is reused. This is the global data page. The **RMUseCnt** field for this page is two; it is one for every other page in use in the system.

Note also that real pages 4 through 6 have been allocated. These pages (along with page 7) are "hidden" behind the HTLB I/O and Boot Space pages number 4 through 7. The Monitor reserved these pages before initialization and then freed them before loading the user program.

Figure 9.13 Initial User Virtual Memory Map

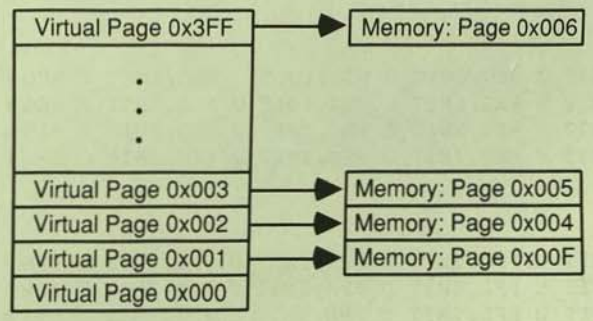
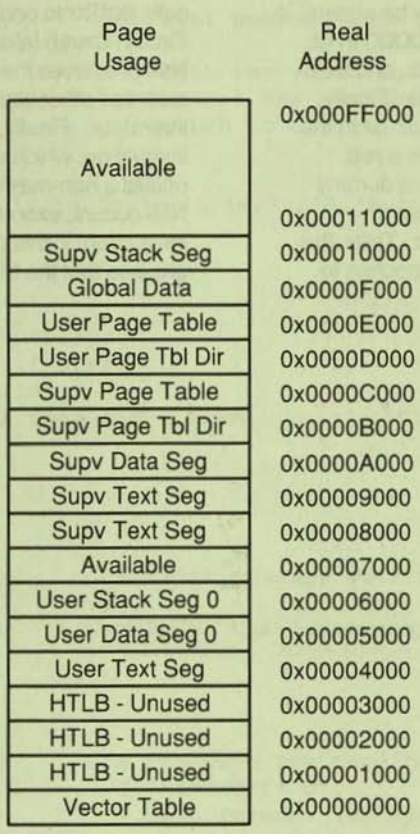


Figure 9.14 illustrates the usage of each allocated real memory page in the system.

Figure 9.14 Real Memory Usage After User Program Loaded



9.4.11. Fatal Error Handlers

This section describes the routines that handle fatal Monitor and user errors. It also describes several support routines called by these handlers.

9.4.11.1 Abort - Terminate User Program

Abort terminates the execution of a user program due to a fatal error condition. **Abort** is always called from a Trap or System Call handler. It expects the address of a diagnostic message in **r0**, a message parameter (which may be absent) in **r1**, and the parameter flag (**PARM_XXXX**) in **r2**. It calls **PrtMsgV** to print the diagnostic, and then calls **PutStr** to print an abort message. Finally, **Abort** reloads the supervisor stack pointer to the value it had in **CSwitch**, and executes a **reti** instruction. Recall that **CSwitch** built a dummy INTRAP stack frame using the address of **CSwExit** before switching to the user. Thus, the **reti** instruction in **Abort** effectively branches to **CSwExit**, restoring the **SSW** and **PSW** in effect before the context switch.

9.4.11.2 Crash - Terminate Monitor Program

Crash terminates the execution of the Monitor program due to a fatal error condition. **Crash** expects the address of a diagnostic message in **r0**, a message parameter (which may be absent) in **r1**, and the parameter flag (**PARM_XXXX**) in **r2**. It calls **PrtMsgV** to print the message and then calls **PutStr** to print a system crash message. Finally, **Crash** falls through to the **Halt** routine. **Halt** preserves the state of the **SSW_M** bit but clears all other bits in the **SSW**, thereby disabling interrupts. Finally, **Halt** issues a **wait** instruction, which effectively halts the CLIPPER unless a non-maskable interrupt is received. If an NMI occurs, execution will resume after the **wait** instruction: in this case, the branch back to **Halt** ensures that the Monitor never leaves the routine.

```
1370: #
1371: # Abort a user program.
1372: # Expects:
1373: #    R0 = Address of message.
1374: #    R1 = Parameter (if present)
1375: #    R2 = Parameter flag
1376: #    R14 = Address of trap stack frame
1377: # Returns:
1378: #    Does not return
1379: #
1380:     .align 2
1381: Abort:
1382:     call   sp,PrtMsgV       # Print message with variable
1383:     loada  _abtmsg,r0
1384:     call   sp,PutStr        # Print the abort message
1385:     loadw  CSwitchSP,sp     # Restore the Monitor's SP
1386:     reti   sp               # Return to CSwitch
1387:
1388: _abtmsg:
1389:     .byte  '*,' ,'P,'r,'o,'g,'r,'a,'m,' ,'A,'b,'o,'r,'t,'e,'d,'.,NL,0
1390:
1391: #
1392: # Crash the system.
1393: # Expects:
1394: #    R0 = Address of message.
1395: #    R1 = Parameter (if present)
1396: #    R2 = Parameter flag
1397: #    R14 = Address of trap stack frame
1398: # Returns:
1399: #    Does not return
1400: #
1401:     .align 2
1402: Crash:
1403:     call   sp,PrtMsgV       # Print message with variable
1404:     loada  _crmsg,r0
1405:     call   sp,PutStr        # Print the crash message
1406:
1407: # Halt the machine.
1408: Halt:
1409:     movpw  ssw,r0
1410:     andi   $SSW_M,r0        # Preserve the mapping state
1411:     movwp  r0,ssw           # Interrupts off
1412:     wait                    # Wait forever (except NMI)
1413:     b      Halt
1414:
1415: _crmsg:
1416:     .byte  '*,' ,'S,'y,'s,'t,'e,'m,' ,'C,'r,'a,'s,'h,' ,NL,NL,0
```

9.4.11.3 PrtMsgV - Print Message with Variable

PrtMsgV prints a message string followed by the hexadecimal representation of the parameter and terminated by a new line. The parameter is either 8-bits long, 32-bits long, or absent entirely.

PrtMsgV expects the address of the message in **r0**, a message parameter (which may be absent) in **r1**, and the parameter flag (**PARM_XXXX**) in **r2**.

It saves the parameter and parameter flag on the stack and calls **PutStr** to print the message. Then, it calls either **PutHexW** or **PutHexB** to print the parameter (if it is present). Finally, it calls **NewLine** to terminate the message.

Note that **PrtMsgV** assumes that messages without a parameter do not need a new line terminator.

```
1417: #
1418: # Print a message that contains a variable.
1419: #
1420: # Expects:
1421: #   R0 = Address of message
1422: #   R1 = Variable
1423: #   R2 = Flag describing variable
1424: # Returns:
1425: #   Nothing
1426: #
1427:     .align 2
1428: PrtMsgV:
1429:     pushw  r1,sp            # Save the parameter
1430:     pushw  r2,sp            #  and the parameter flag
1431:     call   sp,PutStr        # Print the message
1432:     popw   sp,r2            # Restore the parameter flag
1433:     popw   sp,r1            #  and the parameter
1434:     cmpq   $PARM_NULL,r2    # Is there a paramter to print?
1435:     bceq   _pmdone          #  No - done
1436:     movw   r1,r0            # R0 = parameter
1437:     loada  PutHexB,r1       # Assume parameter is a byte
1438:     cmpq   $PARM_BYTE,r2    # Is it?
1439:     bceq   _pmparm          #  Yes - go print it
1440:     loada  PutHexW,r1       #  No - print a word
1441: _pmparm:
1442:     call   sp,(r1)
1443:     call   sp,NewLine
1444: _pmdone:
1445:     ret    sp
```

9.4.11.4 DumpEStk - Dump the Exception Stack Frame

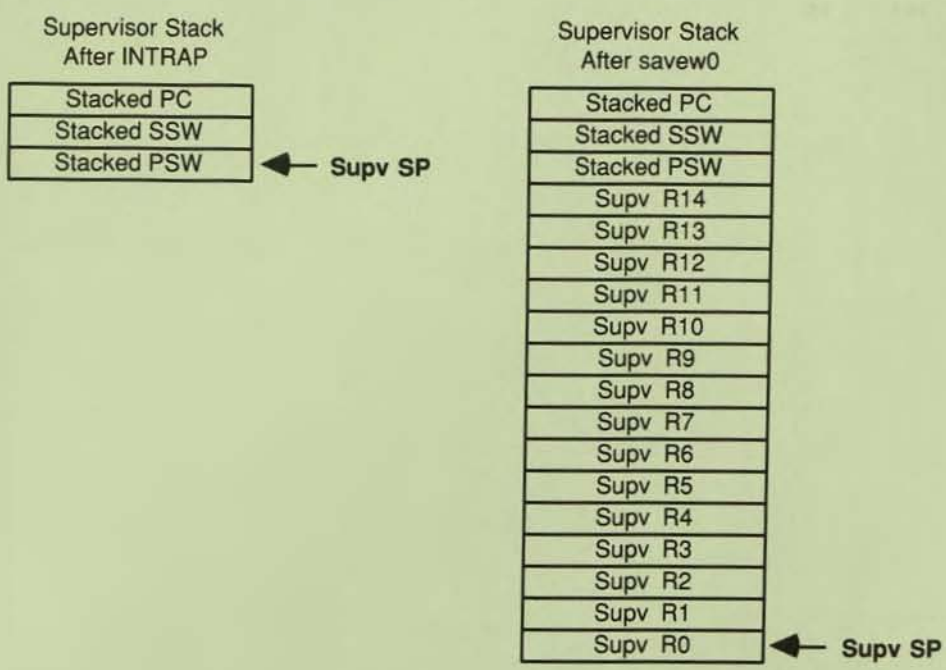
DumpEStk prints a formatted copy of the INTRAP stack frame. It expects **r14** to contain the base address of the frame. It calls **PutStr**, **PutHexW**, and **NewLine** in succession for **PC**, the **SSW**, and the **PSW** register. Note how the Monitor retrieves the value of each of the registers from the stack. Every exception handler places the base address of the INTRAP stack frame into register **r14** immediately after saving the working registers. The lines of code in Example 9.13 are common to all exception handlers.

The **SP** register points to the INTRAP stack frame upon entry to the exception handler. The exception handler always saves the working (supervisor) registers, since the exception might have occurred while the Monitor was executing in supervisor mode. Figure 9.15 shows the supervisor stack immediately after entry to the exception handler, and again, after the execution of the **savew0** instruction.

Example 9.13

```
savew0                          # Save R0 - R14
loada   (15*4)(sp),r14          # R14 = addr of INTRAP stack frame
```

Figure 9.15 Supervisor Stack Within an Exception Handler

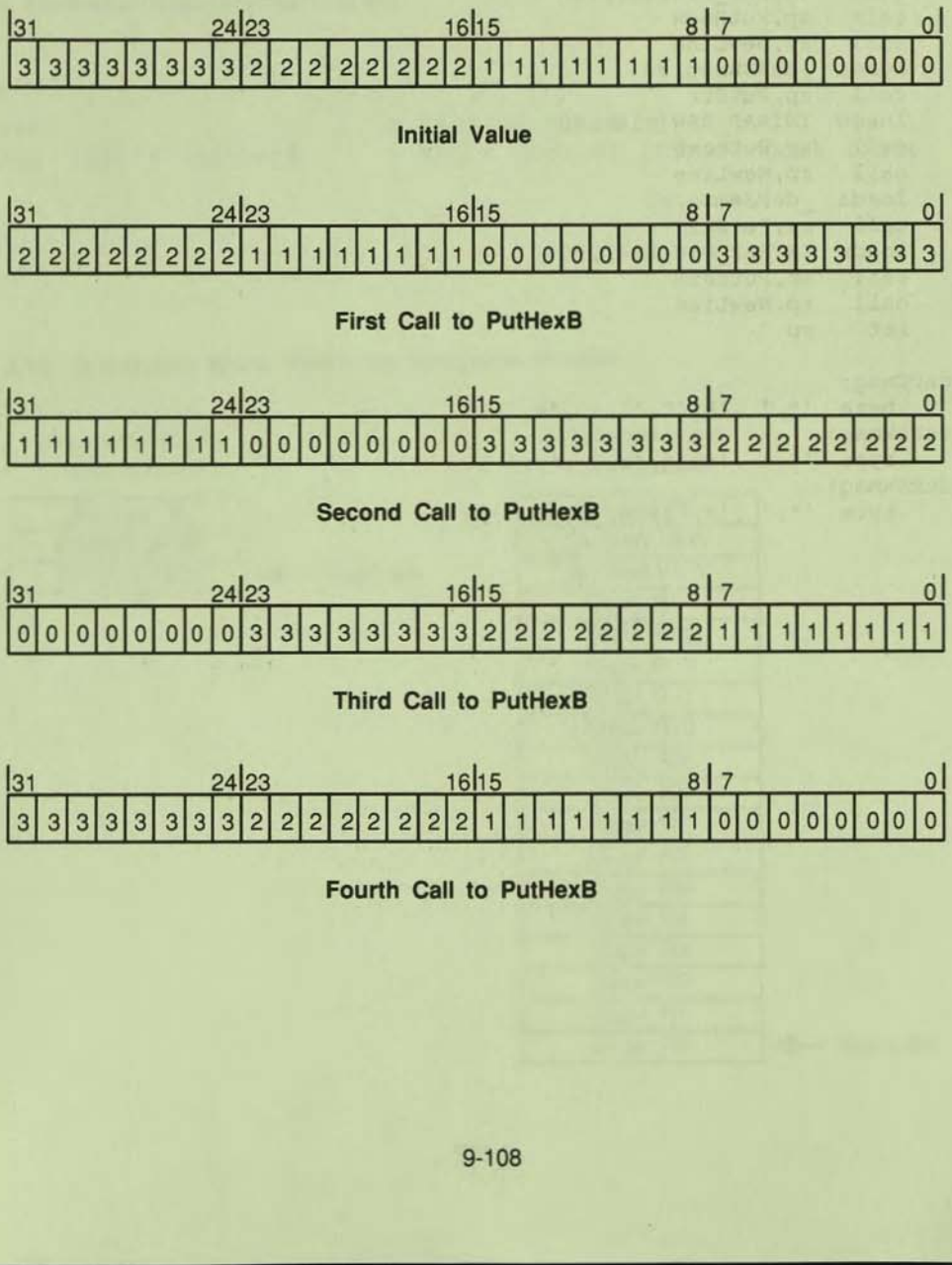


```
1446: #
1447: # Dump the Exception stack frame.
1448: #
1449: # Expects:
1450: #   R14 = address of Trap stack frame
1451: # Returns:
1452: #   Nothing.
1453: #
1454:     .align 2
1455: DumpEStk:
1456:     loada  _dePCmsg,r0
1457:     call   sp,PutStr
1458:     loadw  INTRAP_PC(r14),r0
1459:     call   sp,PutHexW
1460:     call   sp,NewLine
1461:     loada  _deSSWmsg,r0
1462:     call   sp,PutStr
1463:     loadw  INTRAP_SSW(r14),r0
1464:     call   sp,PutHexW
1465:     call   sp,NewLine
1466:     loada  _dePSWmsg,r0
1467:     call   sp,PutStr
1468:     loadw  INTRAP_PSW(r14),r0
1469:     call   sp,PutHexW
1470:     call   sp,NewLine
1471:     ret    sp
1472:
1473: _dePCmsg:
1474:     .byte  '*,' ,'P,'C,' ,' ,'=,' ,0
1475: _deSSWmsg:
1476:     .byte  '*,' ,'S,'S,'W,' ,'=,' ,0
1477: _dePSWmsg:
1478:     .byte  '*,' ,'P,'S,'W,' ,'=,' ,0
```

9.4.11.5 PutHexW - Output 32-bit Hexadecimal Value

PutHexW outputs a 32-bit word to the **SysOut** device in the form of four 8-bit bytes. It calls **PutStr** to output the string "0x" and then falls into **PutHex32** to output the word. **PutHex32** rotates the value so that bytes three, two, one, and finally zero are in the low-order bit positions. After positioning the value, it calls **PutHex8** to output a single byte. Figure 9.16 illustrates the value during each loop iteration.

Figure 9.16 PutHexW R0 Contents



```
1479: #
1480: # Output a 32-bit hexadecimal value to the SysOut device.
1481: # Expects:
1482: #    R0 = value
1483: # Returns:
1484: #    Nothing
1485: #
1486:     .align 2
1487: PutHexW:
1488:     pushw  r0,sp
1489:     loada  _hexmsg,r0
1490:     call   sp,PutStr
1491:     popw   sp,r0
1492:
1493: PutHex32:
1494:     movw   r0,r3            # R3 = value
1495:     loadq  $4,r2            # R2 = count = 4, 8-bit bytes
1496:
1497: _px32loop:
1498:     roti   $8,r3
1499:     movw   r3,r0            # R0 = byte to output
1500:     call   sp,PutHex8
1501:     subq   $1,r2
1502:     bcne   _px32loop
1503:     ret    sp
```

9.4.11.6 PutHexB - Output 8-bit Hexadecimal Value

PutHexB outputs an 8-bit byte to the **SysOut** device in the form of two 4-bit nibbles. It calls **PutStr** to output the string "0x" and then falls into **PutHex8** to output the byte. **PutHex8** shifts the byte so that nibble one and then nibble zero are in the low-order bit positions. After positioning the nibble, **PutHex8** translates the binary value into its ASCII hexadecimal representation and then calls **PutByte** to output the character.

```
1504: #
1505: # Output an 8-bit hexadecimal value to the SysOut device.
1506: # Expects:
1507: #    R0 = value
1508: # Returns:
1509: #    Nothing
1510: #
1511:     .align 2
1512: PutHexB:
1513:     pushw  r0,sp
1514:     loada  _hexmsg,r0
1515:     call   sp,PutStr
1516:     popw   sp,r0
1517:
1518: PutHex8:
1519:     pushw  r0,sp              # Save the value
1520:     shli   $-4,r0
1521:     andi   $0xF,r0            # R0 = high-order nibble
1522:     loadbu _hextbl(r0),r0     # Translate binary value to ASCII char
1523:     call   sp,PutByte
1524:     popw   sp,r0              # R0 = value
1525:     andi   $0xF,r0            # R0 = low-order nibble
1526:     loadbu _hextbl(r0),r0     # Translate binary value to ASCII char
1527:     call   sp,PutByte
1528:     ret    sp
1529:
1530: _hexmsg:
1531:     .byte  '0,'x,0
1532:
1533: _hextbl:
1534:     .byte  '0,'1,'2,'3,'4,'5,'6,'7,'8,'9,'A,'B,'C,'D,'E,'F
```

9.4.12 Trap Handlers

This section describes the handlers for the data memory, floating-point arithmetic, integer arithmetic, instruction memory, illegal operation, and diagnostic trap groups.

9.4.12.1 T_Unimpl - Unimplemented Trap Handler

T_Unimpl process all unimplemented traps. Although unimplemented traps should never occur, good coding practice demands that the Monitor include a trap handler to protect against the possibility. **T_Unimpl** simply loads the address of a diagnostic message into **R0**, extracts the low-order 8 bits from the **SSW**, and branches to **ExCrash**, which dumps the exception stack frame and crashes the system.

The lower 8 bits of the **SSW** contain the number of the trap that was taken. Actually, the **SSW_IL** and **SSW_IN** bits are defined to reside here. However, the Monitor initialized the Vector Table **SSW** entries for all unimplemented traps to have interrupts disabled and to contain the number of the trap in the low-order bits. This trick works as long as the **SSW** value in the Vector Table does not enable interrupts.

```
1535: # **********************
1536: # *                    *
1537: # *   Trap Handlers.   *
1538: # *                    *
1539: # **********************
1540:
1541: #
1542: # Unimplemented Trap.
1543: #
1544:     .align 2
1545: T_Unimpl:
1546:     savew0 # Save R0 - R14
1547:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1548:     loada  _tunmsg,r0       # R0 = message address
1549:     movpw  ssw,r1
1550:     andi   $(SSW_IL|SSW_IN),r1 # R1 = interrupt level & number
1551:     loadq  $PARM_BYTE,r2    # R2 = message parameter is a byte
1552:     b      ExCrash          # Crash the machine
1553:
1554: _tunmsg:
1555:     .byte  '*,' ,'U,'n,'i,'m,'p,'l,'e,'m,'e,'n,'t,'e,'d,' ,'T,'r,'a,'p
1556:     .byte  ' ,'=,' ,0
```

9.4.12.2 T_DCMemErr/T_ICMemErr - Corrected Memory Trap Handler

This routine has two different names, because it handles corrected memory error traps for both data and instruction memory. It saves the working (supervisor) registers, calls **CMemErr** to process the error, restores the registers and returns from the exception.

9.4.12.3 CMemErr - Process Corrected Memory Error

CMemErr processes corrected errors in both data and instruction memory. It can be called from **T_DCMemErr/T_ICMemErr** as a result of a single trap, or from any of the floating-point trap handlers, in the event of a multiple trap. **CMemErr** retrieves the address of the memory error from the memory error address latch (implemented by the memory interface), prints a diagnostic error, and returns.

9.4.12.4 T_DUMemErr/T_IUMemErr - Uncorrectable Memory Trap Handler

This routine has two different names, because it handles uncorrectable memory error traps for both data and instruction memory. It does not save the working (supervisor) registers because uncorrectable memory errors are always fatal in this implementation. Instead, it copies the current stack pointer (which is also the address of the INTRAP stack frame) directly into **r14**, loads the address of the memory error from the memory error address latch (implemented by the memory interface), and branches unconditionally to **ExCrash**, which dumps the exception stack frame and crashes the system.

```
1557: #
1558: # Data and Instruction Memory Corrected Memory Error.
1559: #
1560:     .align 2
1561: T_DCMemErr:
1562: T_ICMemErr:
1563:     savew0 # Save R0 - R14
1564:     loada  (15*4)(sp),r14   # R14 = address of INTRAP stack frame
1565:     call   sp,CMemErr       # Process the error
1566:     restw0                  # Restore R0 - R14
1567:     reti   sp
1568:
1569: #
1570: # Process Corrected memory errors.
1571: # Expects:
1572: # R14 = address of INTRAP stack frame
1573: # Returns:
1574: #   Nothing
1575: #
1576:     .align 2
1577: CMemErr:
1578:     loada  _cmemsg,r0       # R0 = message address
1579:     loadw  SVAD_MERR,r1     # Real memory error address
1580:     loadq  $PARM_LONG,r2    # R2 = message parameter is a long
1581:     call   sp,PrtMsgV       # Print the message
1582:     ret    sp
1583:
1584: _cmemsg:
1585:     .byte  '*,' ,'C,'o,'r,'r,'e,'c,'t,'e,'d
1586:     .byte  ' ,'M,'e,'m,'o,'r,'y,' ,'E,'r,'r,'o,'r,' ,'-
1587:     .byte  ' ,'R,'e,'a,'l,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
1588:
1589: #
1590: # Data and Instruction Memory Uncorrectable Memory Error.
1591: #
1592:     .align 2
1593: T_DUMemErr:
1594: T_IUMemErr:
1595:     movw   sp,r14           # R14 = addr of INTRAP frame on stack
1596:
1597: # Enter here from multiple trap handler.
1598: UMemErr:
1599:     loada  _umemsg,r0       # R0 = message address
1600:     loadw  SVAD_MERR,r1     # Real memory error address
1601:     loadq  $PARM_LONG,r2    # R2 = message parameter is a long
1602:     b      ExCrash          # Crash the system
1603:
1604: _umemsg:
1605:     .byte  '*,' ,'U,'n,'c,'o,'r,'r,'e,'c,'t,'a,'b,'l,'e
1606:     .byte  ' ,'M,'e,'m,'o,'r,'y,' ,'E,'r,'r,'o,'r,' ,'-
1607:     .byte  ' ,'R,'e,'a,'l,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
```

9.4.12.5 T_DPageFlt - Data Page Fault Trap Handler

T_DPageFlt handles data page faults. It saves the working (supervisor) registers, calls **DPageFlt** to process the fault, restores the registers and returns from the exception. Note, however, that some data page faults are fatal. In this case, **DPageFlt** crashes the system without returning.

9.4.12.6 DPageFlt - Process Data Page Fault

DPageFlt processes data page faults. These faults are fatal in supervisor mode unless they occur while the Monitor is accessing user program space: they are fatal in user mode unless they are the result of a stack operation. Whenever the user causes a data page fault by executing an instruction that makes the stack grow, the Monitor allocates another page to the user stack and returns from the exception. Actually, it is very difficult for an operating system to determine exactly what the user was doing when the page fault occurred. This implementation simply extends the stack by one page if the new page will contain the virtual fault address. If the faulting address lies more than one page outside the current stack, the Monitor aborts the user program.

Upon entry, **DPageFlt** checks the **SSW_P** bit to determine if the trap was taken in supervisor or user mode. If the bit is set, the trap occurred in user mode. If the trap occurred in supervisor mode, **DPageFlt** retrieves the **SSW** (that is, the Monitor's **SSW** at the time the trap was taken) from the INTRAP stack frame. If the **SSW_UU** bit is set in the stacked **SSW**, the trap occurred while the Monitor was attempting to access user memory. In this case, the Monitor treats the fault as though it occurred in user mode. If the **SSW_UU** bit is not set in the stacked **SSW**, the Monitor was not attempting to access user space. Thus, the Monitor is faulty, so **DPageFlt** branches to **ExCrash** to print a diagnostic and crash the system.

The code at **_dpfuser** checks to see if the virtual fault address lies within one page of the current lowest stack page. Remember that CLIPPER stacks grow from high to low addresses: the "bottom of stack page" contains the element on the top of the stack. If the virtual fault address is above the current stack, or below the bottom of the next lowest page, **DPageFlt** branches to **_dpfabort** to crash the system.

```
1608: #
1609: # Data Memory Page Fault.
1610: #
1611:     .align 2
1612: T_DPageFlt:
1613:     savew0 # Save R0 - R14
1614:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1615:     call   sp,DPageFlt      # Process the fault
1616:     restw0 # Restore the user's registers
1617:     reti   sp
1618:
1619: #
1620: # Process a data memory page fault.
1621: # Expects:
1622: #   R14 = address of INTRAP stack frame
1623: # Returns:
1624: #   Nothing
1625: #
1626:     .align 2
1627: DPageFlt:
1628:     loadw  C_D_VFA,r1       # R1 = virtual fault address
1629:     movpw  ssw,r3
1630:     andi   $SSW_P,r3        # Was trap taken in user mode?
1631:     brne   _dpfuser         #  Yes - process user mode fault
1632:     loadw  INTRAP_SSW(r14),r3
1633:     andi   $SSW_UU,r3       # Was supv accessing user space?
1634:     brne   _dpfuser         #  Yes - treat it like a user mode page fault
1635:     loada  _dpfmsg,r0       # R0 = message address
1636:     loadq  $PARM_LONG,r2    # R2 = message parameter (VFA) is a long
1637:     b      ExCrash          # Supervisor mode page fault - crash the system
1638:
1639: _dpfuser:                         # Assumes R1 = Virtual Fault Address
1640:     loadhu U_BStkPg,r2      # R2 = lowest page of user's stack seg
1641:     shli   $SHF_VA_PTI,r2   # R2 = virtual address of lowest stack page
1642:     cmpw   r1,r2            # Is the fault address above the current stack?
1643:     bcge   _dpfabort        #  Yes - wild data reference
1644:     subi   $0x00001000,r2   # R2 = virtual address of new stack bottom
1645:     cmpw   r1,r2            # Is the fault address below the new bottom?
1646:     bclt   _dpfabort        #  Yes - requires more than 1 new stack page
```

If control comes to line 1647, the fault occurred when the user program attempted to reference data at or near the top its stack. First, the Monitor gets the user's **PC** address from the INTRAP stack frame and calls **GetUserH** to read the opcode of the instruction that faulted. If the faulting instruction was a **pushw**, the trap handler must adjust the stack pointer upwards by four bytes. In essence, the **pushw** instruction decrements the **SP** and then attempts to write the pushed data to memory. The Monitor would need to adjust the **SP** downward for **popw** instructions when they fault, but in this implementation, faults on pops cause the user program to be aborted.

After adjusting the stack (if necessary), the Monitor attempts to allocate one more real memory page and map it onto the bottom of the stack (remember that CLIPPER stacks grow from high to low memory addresses). First, **DPageFlt** loads the new page usage value (**PGU_USTK**) into **r0** and calls **AllocPage** to allocate a new page of memory. If **AllocPage** fails (returns with the **PSW_C** bit set), **DPageFlt** aborts the user program with an appropriate diagnostic. If a new page was allocated, **DPageFlt** attempts to extend the user's stack segment. First, it loads the current user virtual page number of the stack bottom (**U_BStkPg**), decrements it, and compares it with the current user virtual page number of the data segment top (**U_TDatPg**). If the new stack bottom is less than or equal to the current data top, there is a collision, called a stack bounds violation. **DPageFlt** aborts the user program in this case. Recall that the user can extend the data segment upward by executing a **SYS_EXDS** system call. Since the Monitor can also extend the stack downwards, the possibility of collision exists and must be diagnosed, both here, and in **SC_ExDS**, the **SYS_EXDS** system call handler.

If no collision occurs, **DPageFlt** sets the new bottom of stack page, increments the stack segment length (**U_StkLen**), and calls **SetPte** to map the new page into the user's virtual memory space. **DPageFlt** then returns, and ultimately, the user program is restarted with a larger stack segment.

```
1647:     loadw  INTRAP_PC(r14),r0 # R0 = PC of faulting instruction
1648:     call   sp,GetUserH      # R0 = faulting instruction
1649:     andi   $0xFF00,r0       # R0 = opcode of faulting instruction
1650:     cmpi   $OP_PUSHW,r0     # Was the faulting instruction a PUSHW?
1651:     bcne   _dpfnadj         #  No - don't need to adjust the stack
1652:     movus  sp,r0            #  Yes - get the user's sp
1653:     addq   $4,r0            #    set it to where it was before PUSHW faulted
1654:     movsu  r0,sp            #    and restore it
1655: _dpfnadj:
1656:     loadq  $PGU_STK,r0      # Usage for new page
1657:     call   sp,AllocPage     # R0 = page number
1658:     bc     _dpfnopg         # No memory available
1659:     loadhu U_BStkPg,r1      # R1 = bottom page of user's stack seg
1660:     subq   $1,r1            # R1 = new bottom of stack
1661:     loadhu U_TDataPg,r2     # R2 = top of user's data seg
1662:     cmpw   r1,r2
1663:     bcle   _dpfstkb         # Stack seg <= data seg: stack bounds error
1664:     storh  r1,U_BStkPg      # Remember the new bottom of stack
1665:     loadhu U_StkLen,r2
1666:     addq   $1,r2            # Add 1 page to stack length
1667:     storh  r2,U_StkLen
1668:     loadi  $PTBL_STK,r2
1669:     loada  SVAD_UPT,r3
1670:     call   sp,SetPte        # Set the Page Table Entry for the new stk pg
1671:     ret    sp
1672:
1673: _dpfnopg:
1674:     loada  _dpfsmsg,r0      # R0 = message address
1675:     b      _dpfexit         # Abort the user
1676:
1677: _dpfstkb:
1678:     loada  _dpfbmsg,r0      # R0 = message address
1679:     b      _dpfexit         # Abort the user
1680:
1681: _dpfabort:
1682:     loada  _dpfmsg,r0       # R0 = message address
1683: _dpfexit:
1684:     loadq  $PARM_NULL,r2    # R2 = no message parameter
1685:     b      ExAbort          # Abort the user program
1686:
1687: _dpfmsg:
1688:     .byte  '*,' ,'D,'a,'t,'a,' ,'P,'a,'g,'e,' ,'F,'a,'u,'l,'t,' ,'-
1689:     .byte  ' ,'V,'i,'r,'t,'u,'a,'l,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
1690:
1691: _dpfsmsg:
1692:     .byte  '*,' ,'S,'t,'a,'c,'k,' ,'F,'a,'u,'l,'t,' ,'-,' ,'N,'o
1693:     .byte  ' ,'A,'v,'a,'i,'l,'a,'b,'l,'e,' ,'M,'e,'m,'o,'r,'y,NL,0
1694:
1695: _dpfbmsg:
1696:     .byte  '*,' ,'S,'t,'a,'c,'k,' ,'F,'a,'u,'l,'t,' ,'-,' ,'S,'t,'a,'c,'k
1697:     .byte  ' ,'a,'n,'d,' ,'D,'a,'t,'a,' ,'C,'o,'l,'l,'i,'s,'i,'o,'n,'.,NL,0
```

9.4.12.7 T_DRdProt - Data Memory Read Protection Fault Trap Handler

T_DRdProt processes read protection faults, which are always fatal in this implementation. The handler loads the address of the diagnostic message and reads the D-CAMMU's virtual fault address register (**C_VFA**) to determine the fault address. It then checks the **SSW_P** bit to determine if the fault was taken in user or supervisor mode. If the fault was caused by the user, the **SSW_P** bit will be set. If it is a user fault, **T_DRdProt** branches to **Abort** to terminate the user program. If it is a supervisor fault, **T_DRdProt** fetches the **SSW** register from the INTRAP stack frame and checks the state of the **SSW_UU** bit. If it is set, the trap occurred as the result of a supervisor read from user memory during system call processing. In this case also, **T_DRdProt** aborts the user program.

If the **SSW_UU** bit is not set, it is a supervisor fault. In this case, **T_RdProt** branches unconditionally to **ExCrash**, which dumps the exception stack frame and crashes the system.

```
1698: #
1699: # Data Memory Read Protection Fault.
1700: #
1701:     .align 4
1702: T_DRdProt:
1703:     savew0 # Save R0 - R14
1704:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1705:
1706: # Enter here from multiple trap handler.
1707: DRdProt:
1708:     loada  _rdprmsg,r0      # R0 = message address
1709:     loadw  C_D_VFA,r1       # R1 = virtual fault address
1710:     loadq  $PARM_LONG,r2    # R2 = message parameter is a long
1711:     movpw  ssw,r3
1712:     andi   $SSW_P,r3        # Was trap taken in User mode?
1713:     brne   ExAbort          #  Yes - abort the user program
1714:     loadw  INTRAP_SSW(r14),r3
1715:     andi   $SSW_UU,r3       # Was it a Supv access to user space?
1716:     brne   ExAbort          #  Yes - abort the user program
1717:     b      ExCrash          #  No - crash the system
1718:
1719: _rdprmsg:
1720:     .byte  '*,' ,'R,'e,'a,'d,' ,'P,'r,'o,'t,'e,'c,'t,'i,'o,'n
1721:     .byte  ' ,'F,'a,'u,'l,'t,' ,'-
1722:     .byte  ' ,'V,'i,'r,'t,'u,'a,'l,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
```

9.4.12.8 T_DWrtProt - Data Memory Write Protection Fault Trap Handler

T_DWrtProt processes write protection faults, which are always fatal in this implementation. The handler loads the address of the diagnostic message and reads the D-CAMMU's virtual fault address register (**C_VFA**) to determine the fault address. It then checks the **SSW_P** bit to determine if the fault was taken in user or supervisor mode. If the fault was caused by the user, the **SSW_P** bit will be set. If it is a user fault, **T_DWrtProt** branches to **Abort** to terminate the user program. If it is a supervisor fault, **T_DWrtProt** fetches the **SSW** register from the INTRAP stack frame and checks the state of the **SSW_UU** bit. If it is set, the trap occurred as the result of a supervisor write to user memory during system call processing. In this case also, **T_DWrtProt** aborts the user program.

If the **SSW_UU** bit is not set, it is a supervisor fault. In this case, **T_RdProt** branches unconditionally to **ExCrash**, which dumps the exception stack frame and crashes the system.

```
1723: #
1724: # Data Memory Write Protection Fault.
1725: #
1726:     .align 4
1727: T_DWrtProt:
1728:     savew0 # Save R0 - R14
1729:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1730:
1731: # Enter here from multiple trap handler.
1732: DWrtProt:
1733:     loada  _wrtprmsg,r0     # R0 = message address
1734:     loadw  C_D_VFA,r1       # R1 = virtual fault address
1735:     loadq  $PARM_LONG,r2    # R2 = message parameter is a long
1736:     movpw  ssw,r3
1737:     andi   $SSW_P,r3        # Was trap taken in User mode?
1738:     brne   ExAbort          #  Yes - abort the user program
1739:     loadw  INTRAP_SSW(r14),r3
1740:     andi   $SSW_UU,r3       # Was it a Supv access to user space?
1741:     brne   ExAbort          #  Yes - abort the user program
1742:     b      ExCrash          #  No - crash the system
1743:
1744: _wrtprmsg:
1745:     .byte  '*,' ,'W,'r,'i,'t,'e,' ,'P,'r,'o,'t,'e,'c,'t,'i,'o,'n
1746:     .byte  ' ,'F,'a,'u,'l,'t,' ,'-
1747:     .byte  ' ,'V,'i,'r,'t,'u,'a,'l,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
```

9.4.12.9 T_FPInex - Floating-Point Inexact Trap Handler

T_FPInex processes floating-point inexact traps, which are always fatal in this implementation. After saving the (supervisor's) general registers, **T_FPInex** executes a **loadfs** instruction to determine the address of the faulting instruction and saves the **PC** value in **FPTrapPC**. Then, it calls **ChkMTrap** to check for and process a simultaneous memory data trap. When (and if) **ChkMTrap** returns, **T_FPInex** loads the diagnostic message parameters and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

9.4.12.10 T_FPUflow - Floating-Point Underflow Trap Handler

T_FPUflow processes floating-point underflow traps, which are always fatal in this implementation. After saving the (supervisor's) general registers, **T_FPUflow** executes a **loadfs** instruction to determine the address of the faulting instruction and saves the **PC** value in **FPTrapPC**. Then, it calls **ChkMTrap** to check for and process a simultaneous memory data trap. When (and if) **ChkMTrap** returns, **T_FPUflow** loads the diagnostic message parameters and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

```
1748: #
1749: # Floating Point Inexact Value.
1750: #
1751:     .align 4
1752: T_FPInex:
1753:     savew0 # Save R0 - R14
1754:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1755:     loadfs r0,f0            # R0=PC of floating point instruction
1756:     storw  r0,FPTrapPC      # Save it
1757:     call   sp,ChkMTrap      # Check for and process data memory trap
1758:     loada  _fpimsg,r0       # R0 = message address
1759:     loadw  FPTrapPC,r1      # PC of floating point instruction
1760:     loadq  $PARM_LONG,r2    # Message has no parameter
1761:     b      ExFatal          # Error is fatal to user or supervisor
1762:
1763: _fpimsg:
1764:     .byte  '*,' ,'F,'l,'o,'a,'t,'i,'n,'g,' ,'P,'o,'i,'n,'t
1765:     .byte  ' ,'I,'n,'e,'x,'a,'c,'t,' ,'V,'a,'l,'u,'e
1766:     .byte  ' ,'-,' ,'P,'C,' ,'=,' ,0
1767:
1768: #
1769: # Floating Point Underflow.
1770: #
1771:     .align 4
1772: T_FPUflow:
1773:     savew0 # Save R0 - R14
1774:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1775:     loadfs r0,f0            # R0=PC of floating point instruction
1776:     storw  r0,FPTrapPC      # Save it
1777:     call   sp,ChkMTrap      # Check for and process data memory trap
1778:     loada  _fpumsg,r0       # R0 = message address
1779:     loadw  FPTrapPC,r1      # PC of floating point instruction
1780:     loadq  $PARM_LONG,r2    # Message has no parameter
1781:     b      ExFatal          # Error is fatal to user or supervisor
1782:
1783: _fpumsg:
1784:     .byte  '*,' ,'F,'l,'o,'a,'t,'i,'n,'g,' ,'P,'o,'i,'n,'t
1785:     .byte  ' ,'U,'n,'d,'e,'r,'f,'l,'o,'w,' ,'-,' ,'P,'C,' ,'=,' ,0
```

9.4.12.11 T_FPDiv0 - Floating-Point Divide by Zero Trap Handler

T_FPDiv0 processes floating-point overflow traps, which are always fatal in this implementation. After saving the (supervisor's) general registers, **T_FPDiv0** executes a **loadfs** instruction to determine the address of the faulting instruction and saves the **PC** value in **FPTrapPC**. Then, it calls **ChkMTrap** to check for and process a simultaneous memory data trap. When (and if) **ChkMTrap** returns, **T_FPDiv0** loads the diagnostic message parameters and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

9.4.12.12 T_FPOflow - Floating-Point Overflow Trap Handler

T_FPOflow processes floating-point overflow traps, which are always fatal in this implementation. After saving the (supervisor's) general registers, **T_FPOflow** executes a **loadfs** instruction to determine the address of the faulting instruction and saves the **PC** value in **FPTrapPC**. Then, it calls **ChkMTrap** to check for and process a simultaneous memory data trap. When (and if) **ChkMTrap** returns, **T_FPOflow** loads the diagnostic message parameters and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

```
1786: #
1787: # Floating Point Divide By Zero.
1788: #
1789:     .align 4
1790: T_FPDiv0:
1791:     savew0 # Save R0 - R14
1792:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1793:     loadfs r0,f0            # R0=PC of floating point instruction
1794:     storw  r0,FPTrapPC      # Save it
1795:     call   sp,ChkMTrap      # Check for and process data memory trap
1796:     loada  _fpd0msg,r0      # R0 = message address
1797:     loadw  FPTrapPC,r1      # PC of floating point instruction
1798:     loadq  $PARM_LONG,r2    # Message has no parameter
1799:     b      ExFatal          # Error is fatal to user or supervisor
1800:
1801: _fpd0msg:
1802:     .byte  '*,' ,'F,'l,'o,'a,'t,'i,'n,'g,' ,'P,'o,'i,'n,'t
1803:     .byte  ' ,'D,'i,'v,'i,'d,'e,' ,'B,'y,' ,'Z,'e,'r,'o
1804:     .byte  ' ,'-,' ,'P,'C,' ,'=,' ,0
1805:
1806: #
1807: # Floating Point Overflow.
1808: #
1809:     .align 4
1810: T_FPOflow:
1811:     savew0 # Save R0 - R14
1812:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1813:     loadfs r0,f0            # R0=PC of floating point instruction
1814:     storw  r0,FPTrapPC      # Save it
1815:     call   sp,ChkMTrap      # Check for and process data memory trap
1816:     loada  _fpomsg,r0       # R0 = message address
1817:     loadw  FPTrapPC,r1      # PC of floating point instruction
1818:     loadq  $PARM_LONG,r2    # Message has no parameter
1819:     b      ExFatal          # Error is fatal to user or supervisor
1820:
1821: _fpomsg:
1822:     .byte  '*,' ,'F,'l,'o,'a,'t,'i,'n,'g,' ,'P,'o,'i,'n,'t
1823:     .byte  ' ,'O,'v,'e,'r,'f,'l,'o,'w,' ,'-,' ,'P,'C,' ,'=,' ,0
```

9.4.12.13 T_FPInvOp - Floating-Point Invalid Operation Trap Handler

T_FPInvOp processes floating-point invalid operation traps, which are always fatal in this implementation. After saving the (supervisor's) general registers, **T_FPInvOp** executes a **loadfs** instruction to determine the address of the faulting instruction and saves the **PC** value in **FPTrapPC**. Then, it calls **ChkMTrap** to check for and process a simultaneous memory data trap. When (and if) **ChkMTrap** returns, **T_FPInvOp** loads the diagnostic message parameters and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

9.4.12.14 T_IDiv0 - Integer Divide by Zero Trap Handler

T_IDiv0 processes integer divide by zero traps, which are always fatal in this implementation. It loads the address of the diagnostic message and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

```
1824: #
1825: # Floating Point Invalid Operation.
1826: #
1827:     .align 2
1828: T_FPInvOp:
1829:     savew0 # Save R0 - R14
1830:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1831:     loadfs r0,f0            # R0=PC of floating point instruction
1832:     storw  r0,FPTrapPC      # Save it
1833:     call   sp,ChkMTrap      # Check for and process data memory trap
1834:     loada  _fpiomsg,r0      # R0 = message address
1835:     loadw  FPTrapPC,r1      # PC of floating point instruction
1836:     loadq  $PARM_LONG,r2    # Message has no parameter
1837:     b      ExFatal          # Error is fatal to user or supervisor
1838:
1839: _fpiomsg:
1840:     .byte  '*,' ,'F,'l,'o,'a,'t,'i,'n,'g,' ,'P,'o,'i,'n,'t
1841:     .byte  ' ,'I,'n,'v,'a,'l,'i,'d,' ,'O,'p,'e,'r,'a,'t,'i,'o,'n
1842:     .byte  ' ,'-,' ,'P,'C,' ,'=,' ,0
1843:
1844: #
1845: # Integer divide By Zero.
1846: #
1847:     .align 2
1848: T_IDiv0:
1849:     movw   sp,r14           # R14 = addr of INTRAP frame on stack
1850:     loada  _id0msg,r0       # R0 = message address
1851:     loadq  $PARM_NULL,r2    # R2 = message parameter is a long
1852:     b      ExFatal          # The error is fatal to User or Supv
1853:
1854: _id0msg:
1855:     .byte  '*,' ,'I,'n,'t,'e,'g,'e,'r,' ,'D,'i,'v,'i,'d,'e,' ,'b,'y
1856:     .byte  ' ,'Z,'e,'r,'o,'.,NL,0
```

9.4.12.15 T_IPageFlt - Instruction Memory Page Fault Trap Handler

T_IPageFlt processes instruction memory page faults, which are always fatal in this implementation. It loads the address of the diagnostic message and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

9.4.12.16 T_IExProt -Instruction Execution Protection Trap Handler

T_IExProt processes instruction memory execution protection faults, which are always fatal in this implementation. It loads the address of the diagnostic message and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode.

```
1857: #
1858: # Instruction Memory Page Fault.
1859: #
1860:     .align 2
1861: T_IPageFlt:
1862:     movw   sp,r14           # R14 = addr of INTRAP frame on stack
1863:     loada  _ipfmsg,r0       # R0 = message address
1864:     loadw  C_I_VFA,r1       # R1 = virtual fault address
1865:     loadq  $PARM_LONG,r2    # R2 = message parameter is a long
1866:     b      ExFatal          # The error is fatal to User or Supv
1867:
1868: _ipfmsg:
1869:     .byte  '*,' ,'I,'n,'s,'t,'u,'c,'t,'i,'o,'n,' ,'P,'a,'g,'e
1870:     .byte  ' ,'F,'a,'u,'l,'t,' ,'-
1871:     .byte  ' ,'V,'i,'r,'t,'u,'a,'l,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
1872:
1873:
1874: #
1875: # Instruction Memory Execution Protection Fault.
1876: #
1877:     .align 2
1878: T_IExProt:
1879:     movw   sp,r14           # R14 = addr of INTRAP frame on stack
1880:     loada  _exprmsg,r0      # R0 = message address
1881:     loadw  C_I_VFA,r1       # R1 = virtual fault address
1882:     loadq  $PARM_LONG,r2    # R2 = message parameter is a long
1883:     b      ExFatal          # The error is fatal to User or Supv
1884:
1885: _exprmsg:
1886:     .byte  '*,' ,'E,'x,'e,'c,'u,'t,'i,'o,'n
1887:     .byte  ' ,'P,'r,'o,'t,'e,'c,'t,'i,'o,'n,' ,'F,'a,'u,'l,'t,' ,'-
1888:     .byte  ' ,'V,'i,'r,'t,'u,'a,'l,' ,'A,'d,'d,'r,'e,'s,'s,' ,'=,' ,0
```

9.4.12.17 T_IllOp - Illegal Operation Trap Handler

T_IllOp handles illegal operation traps, which are always fatal in this implementation. It loads the address of the illegal instruction from the INTRAP stack frame and then tests the state of the **SSW_P** bit. If the illegal instruction was executed by the supervisor, **T_IllOp** loads the illegal opcode from supervisor virtual memory. If the fault occurred in user mode, **T_IllOp** calls **GetUserH** to fetch the illegal opcode from user virtual memory. In either case, **T_IllOp** then checks to see if the opcode was a **trapfn** instruction. If it was, **T_IllOp** loads the address of the floating unordered diagnostic message and branches unconditionally to **ExFatal**, which aborts the user program or crashes the system, depending on whether the trap was taken in user or supervisor mode. If the illegal operation was not a **trapfn** instruction, **T_IllOp** loads the address of the illegal instruction diagnostic message and exits through **ExFatal**, which either aborts the user program or crashes the system.

```
1889: #
1890: # Illegal Operation.
1891: #
1892:     .align 2
1893: T_IllOp:
1894:     savew0
1895:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1896:     loadw  INTRAP_PC(r14),r0 # R0 = PC of bad instruction
1897:     movpw  ssw,r1           # R1 = current SSW
1898:     andi   $SSW_P,r1        # Was the trap taken in user mode?
1899:     brne   _tiouser         #  Yes
1900:     loadhu (r0),r0          # R0 = faulting instruction
1901:     b      _tiotest
1902:
1903: _tiouser:
1904:     call   sp,GetUserH      # R0 = faulting instruction
1905: _tiotest:
1906:     cmpi   $OP_TRAPFN,r0    # Was it a TRAPFN instruction?
1907:     breq   _tiotfn          #  Yes
1908:     loada  _iomsg,r0        # R0 = message address
1909: _tiodone:
1910:     loadq  $PARM_NULL,r2    # R2 = message parameter is a long
1911:     b      ExFatal          # The error is fatal to User or Supv
1912:
1913: _tiotfn:
1914:     loada  _tfnmsg,r0       # R0 = message address
1915:     b      _tiodone
1916:
1917: _iomsg:
1918:     .byte  '*,' ,'I,'l,'l,'e,'g,'a,'l,' ,'O,'p,'e,'r,'a,'t,'i,'o,'n,'.,NL,0
1919:
1920: _tfnmsg:
1921:     .byte  '*,' ,'F,'l,'o,'a,'t,'i,'n,'g,' ,'U,'n,'o,'r,'d,'e,'r,'e,'d
1922:     .byte  ' ,'C,'o,'n,'d,'i,'t,'i,'o,'n,'.,NL,0
```

9.4.12.18 T_PrivIns - Privileged Instruction Trap Handler

T_PrivIns processes privileged instruction faults, which are always fatal in this implementation. It loads the address of the diagnostic message and branches unconditionally to **ExAbort**, which aborts the user program.

9.4.12.19 T_Trace - Trace Trap Handler

T_Trace processes trace traps, which are always allowed in this implementation. It prints the trace trap message and then dumps the exception stack frame on the **SysOut** device. It then retrieves the stacked **SSW**, clears the trace pending bit (**SSW_TP**), and exits from the exception.

```
1923: #
1924: # Privileged Instruction.
1925: #
1926:     .align 2
1927: T_PrivIns:
1928:     savew0 # Save R0 - R14
1929:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1930:     loada  _pimsg,r0        # R0 = message address
1931:     loadq  $PARM_NULL,r2    # R2 = message parameter is a long
1932:     b      ExAbort          # Abort the user
1933:
1934: _pimsg:
1935:     .byte  '*,' ,'P,'r,'i,'v,'i,'l,'e,'g,'e,'d
1936:     .byte  ' ,'I,'n,'s,'t,'u,'c,'t,'i,'o,'n,'.,NL,0
1937:
1938: #
1939: # Trace Trap.
1940: #
1941:     .align 2
1942: T_Trace:
1943:     savew0 # Save R0 - R14
1944:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1945:     loada  _ttmsg,r0        # R0 = message address
1946:     call   sp,PutStr
1947:     call   sp,DumpEStk      # Dump the exception stack frame
1948:     loadw  INTRAP_SSW(r14),r0
1949:     andi   $~SSW_TP,r0      # Turn off the trace pending bit
1950:     storw  r0,INTRAP_SSW(r14)
1951:     restw0 # Restore R0 - R14
1952:     reti   sp               # Exit the trap
1953:
1954: _ttmsg:
1955:     .byte  '*,' ,'T,'r,'a,'c,'e,' ,'T,'r,'a,'p,'.,NL,0
```

9.4.13 System Call Handlers

This section describes the system call handlers provided by the supervisor program. These routines allow a user program to read and write the **SysIn/SysOut** devices, allocate additional pages of data memory, and terminate processing.

9.4.13.1 SC_Unsupp - Unsupported System Call Handler

Since the user is free to issue the **calls** instruction with any of the 128 possible request codes, the supervisor must have a handler present for each vector. **SC_Unsupp** is a single handler that processes all unsupported system calls. **SC_Unsupp** simply loads the address of a diagnostic message into **R0**, extracts the low-order 8-bits from the **SSW**, and branches to **ExCrash**, which dumps the exception stack frame and crashes the system.

The lower 8 bits of the **SSW** contain the number of the trap that was taken. Actually, the **SSW_IL** and **SSW_IN** bits are defined to reside here. However, the supervisor initialized the Vector Table **SSW** entries for all unsupported system calls to have interrupts disabled and to contain the number of the system call in the low-order bits. This is a trick that works as long as the **SSW** value does not enable interrupts.

9.4.13.2 SC_Exit - SYS_EXIT System Call Handler

SC_Exit processes a user request to terminate the user program. It expects an exit status code in the user's **R0** register. **SC_Exit** calls **PrtMsgV** to print the exit message, loads the stack pointer with the value that was in effect at the time of the context switch (**CSwitchSP**), and performs a **reti** instruction. This returns control to **CSwExit** in supervisor mode, with the **SSW** and **PSW** restored to the values they had before the context switch.

```
1956: # ****************************
1957: # *                          *
1958: # *   System Call Handlers.  *
1959: # *                          *
1960: # ****************************
1961:
1962: #
1963: # Unsupported System Call.
1964: #
1965:     .align 2
1966: SC_Unsupp:
1967:     savew0                  # Save R0 - R14
1968:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
1969:
1970:     loada  _SCUnMsg,r0      # R0 = message address
1971:     movpw  ssw,r1
1972:     andi   $(SSW_IL|SSW_IN),r1 # R1 = interrupt level & number
1973:     loadq  $PARM_BYTE,r2    # R2 = Crash mesage parameter is a byte
1974:     b      ExCrash          # Crash the machine
1975:
1976: _SCUnMsg:
1977:     .byte  '*,' ,'U,'n,'s,'u,'p,'p,'o,'r,'t,'e,'d,' ,'S,'y,'s,'t,'e,'m
1978:     .byte  ' ,'C,'a,'l,'l,' ,'=,' ,0
1979:
1980: #
1981: # System Call #0x00 - exit user program.
1982: # Expects:
1983: #   U0 = exit status
1984: # Returns:
1985: #   Does not return
1986: #
1987:     .align 2
1988: SC_Exit:
1989:     loada  _exmsg,r0        # Message address
1990:     movus  r0,r1            # R1 = exit status
1991:     loadq  $PARM_BYTE,r2    # Message parameter is a byte
1992:     call   sp,PrtMsgV
1993:     loadw  CSwitchSP,sp     # Restore the CSwitch stack pointer
1994:     reti   sp               # Return to CSwitch in supervisor mode
1995:
1996: _exmsg:
1997:     .byte  '*,' ,'P,'r,'o,'g,'r,'a,'m,' ,'E,'x,'i,'t
1998:     .byte  ' ,'-,' ,'S,'t,'a,'t,'u,'s,' ,'C,'o,'d,'e,' ,'=,' ,0
```

9.4.13.3 SC_ExDS - SYS_EXDS System Call Handler

SC_ExDS processes a user request to extend the data segment. After saving the context, **SC_ExDS** attempts to allocate a page of real memory by calling **AllocPage** with the **PGU_UDAT** page usage value. If **AllocPage** fails (returns with the **PSW_C** bit set), **SC_ExDS** exits through **_extfail**, which loads a zero into user register **r0**, restores the context, and returns.

If **AllocPage** succeeds, **SC_ExDS** loads and increments the number of the top page of the user's data segment (**U_TDataPg**), then loads the number of the bottom page of the user's stack segment (**U_BStkPg**) and checks for a collision. This is possible because the data segment is growing upwards, while the stack is growing downwards. If the segments collide, **SC_ExDS** branches to **_extcoll**, which calls **FreeRM** to return the newly allocated page, and then exits through **_extdone**.

If the data and stack segments do not collide, **SC_ExDS** remembers the new top page number of the data segment (**U_TDataPg**), increments the number of pages in the data segment (**U_TDataLen**), and then calls **SetPte** to map the new page into the user's virtual address space. Finally, **SC_ExDs** reloads the number of the top data segment page (this value is a virtual page number) and converts it into a virtual address. **SC_ExDS** returns this address to the user in the **r0** register. A value of zero in this register indicates that the request failed.

```
1999: #
2000: # System Call #0x01 - extend data segment.
2001: # Expects:
2002: #    Nothing
2003: # Returns:
2004: #    U0 = virtual address of base of page if success
2005: #    U0 = 0; no memory available or data and stack segment collision
2006: #
2007:      .align 2
2008: SC_ExDS:
2009:      savew0                  # Save R0 - R14
2010:      loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
2011:      loadq  $PGU_DATA,r0     # Usage for new page
2012:      call   sp,AllocPage     # R0 = real page number
2013:      bc     _extfail         # No memory available
2014:      loadhu U_TDataPg,r1     # R1 = top page of user's data seg
2015:      addq   $1,r1            # R1 = new top of data
2016:      loadhu U_BStkPg,r2      # R2 = bottom of user's stack seg
2017:      cmpw   r1,r2
2018:      bcge   _extfail         # Data seg >= stack seg: failure
2019:      storh  r1,U_TDataPg     # Remember the new top of data
2020:      loadhu U_DataLen,r2
2021:      addq   $1,r2            # Add 1 page to data length
2022:      storh  r2,U_DataLen
2023:      loadi  $PTBL_DATA,r2
2024:      loada  SVAD_UPT,r3
2025:      call   sp,SetPte        # Set the Page Table Entry for the new stk pg
2026:      loadhu U_TDataPg,r0     # R0 = new virtual page number
2027:      shli   $SHF_VA_PTI,r0   # R0 = virtual address of new page
2028: _extdone:
2029:      movsu  r0,r0            # U0 = return value
2030:      restw0                  # Restore the user's registers
2031:      reti   sp
2032:
2033: # No memory available or data and stack segment collision.
2034: _extfail:
2035:      loadq  $0,r0            # R0 = 0 - failure
2036:      b      _extdone
```

9.4.13.4 SC_GetB - SYS_GETB System Call Handler

SC_GetB inputs a byte from the **SysIn** device and returns it to the user in (user) register **R0**. **SC_GetB** saves the context, calls **GetByte** to get the data, moves it into user register **R0**, restores the context, and returns.

9.4.13.5 SC_PutB - SYS_PUTB System Call Handler

SC_PutB fetches the byte in the user's **R0** register and outputs it to the **SysOut** device. **SC_PutB** saves the context, fetches the data from the user's **R0** register, and calls **PutByte** to write the data. Then, **SC_PutB** restores the context, and returns.

```
2037: #
2038: # System Call #0x02 - get byte from SysIn.
2039: # Expects:
2040: #   Nothing
2041: # Returns:
2042: #   U0 - byte from SysIn
2043: #
2044:     .align 2
2045: SC_GetB:
2046:     savew0                  # Save R0 - R14
2047:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
2048:     call   sp,GetByte       # R0 = byte from SysIn
2049:     movsu  r0,r0            # Put the byte in the user's R0
2050:     restw0                  # Restore R0 - R14
2051:     reti   sp               # Exit
2052:
2053: #
2054: # System Call #0x03 - put byte to SysOut.
2055: # Expects:
2056: #   U0 - byte to output
2057: # Returns:
2058: #   Nothing
2059: #
2060:     .align 2
2061: SC_PutB:
2062:     savew0                  # Save R0 - R14
2063:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
2064:     movus  r0,r0            # Get the byte in the user's R0
2065:     call   sp,PutByte       # Output it to SysOut
2066:     restw0                  # Restore R0 - R14
2067:     reti   sp               # Exit
```

9.4.13.6 SC_PutS - SYS_PUTS System Call Handler

SC_PutS outputs a null-terminated string in user virtual memory to the **SysOut** device. It expects the address of the string in user register **R0**. **SC_PutS** saves the context, fetches the address of the string from the user's **R0** register, and enters the character output loop. For each loop iteration, **SC_PutS** calls **GetUserB** to load the next character in the string from user virtual memory. If the character is zero, **SC_PutS** exits from the loop. Note that the code properly handles the degenerate case of a zero-length string.

If the character is non-zero, **SC_PutS** calls **PutByte** to write it to the **SysOut** device, increments the string pointer, and loops for the next character.

```
2068: #
2069: # System Call #0x03 - put null-terminated string to SysOut.
2070: # Expects:
2071: #   U0 = address of string in user space
2072: # Returns:
2073: #   Nothing
2074: #
2075:     .align 2
2076: SC_PutS:
2077:     savew0                  # Save R0 - R14
2078:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
2079:     movus  r0,r0            # R0 = address of string to output
2080: _scpsloop:
2081:     pushw  r0,sp            # Save the user address
2082:     call   sp,GetUserB      # R0 = next byte
2083:     cmpq   $0,r0            # String terminator?
2084:     bceq   _scpsdone        #  Yes - all done
2085:     call   sp,PutByte       # Output the byte
2086:     popw   sp,r0            # R0 = address of current byte
2087:     addq   $1,r0            # R0 = address of next byte in string
2088:     b      _scpsloop        # Do another byte
2089:
2090: _scpsdone:
2091:     popw   sp,r0            # discard the address
2092:     restw0                  # Restore R0 - R14
2093:     reti   sp               # Exit
```

9.4.14 Interrupt Handlers

This section explains the Monitor's interrupt handlers. Only two interrupts are recognized by the system: the non-maskable interrupt, which signifies a power failure in this implementation, and the clock interrupt, which occurs **CLK_FREQ** times per second.

9.4.14.1 I_Unexp - Unexpected Interrupt Handler

Even though there should never be an unexpected interrupt in the system, good coding practice demands that the Monitor have a handler for each of the 256 possible interrupts. **I_Unexp** is a single handler that processes all unexpected interrupts. **I_Unexp** saves the context, loads the address of a diagnostic into **r0**, extracts the low-order 8-bits from the **SSW**, and calls **PrtMsgV** to print the message. Then, **I_Unexp** branches to **ExCrash** to crash the system.

The lower 8 bits of the **SSW** contain the number of the interrupt that was acknowledged. Actually, the **SSW_IL** and **SSW_IN** bits are defined to reside here. However, the Monitor initialized the Vector Table **SSW** entries for all unexpected interrupts to have interrupts disabled and to contain the number of the interrupt in the low-order bits. This is a trick that works as long as the **SSW** value does not enable interrupts.

9.4.14.2 I_Nmi - Non-Maskable Interrupt Handler

A non-maskable interrupt occurs as the result of an impending power failure in this implementation. **I_Nmi** loads the address of a diagnostic into **r0**, gets the current time, and branches unconditionally to **Crash**, which crashes the system.

```
2094: # ****************************
2095: # *                          *
2096: # *   Interrupt Handlers.    *
2097: # *                          *
2098: # ****************************
2099:
2100: #
2101: # Unexpected Interrupt.
2102: #
2103:     .align 2
2104: I_Unexp:
2105:     savew0                  # Save R0 - R14
2106:     loada  (15*4)(sp),r14   # R14 = addr of INTRAP frame on stack
2107:
2108:     loada  _IUnMsg,r0       # R0 = message address
2109:     movpw  ssw,r1
2110:     andi   $(SSW_IL|SSW_IN),r1 # R1 = interrupt level & number
2111:     loadq  $PARM_BYTE,r2    # R2 = Crash mesage parameter is a byte
2112:     b      ExCrash          # Crash the machine
2113:
2114: _IUnMsg:
2115:     .byte  '*,' ,'U,'n,'e,'x,'p,'e,'c,'t,'e,'d
2116:     .byte  ' ,'I,'n,'t,'e,'r,'r,'u,'p,'t,' ,'=,' ,0
2117:
2118: #
2119: # Non-Maskable Interrupt.  Signifies power failure.
2120: #
2121:     .align 2
2122: I_Nmi:
2123:     movw   sp,r14           # R14 = addr of INTRAP frame on stack
2124:     loada  _pfmsg,r0        # R0 = message address
2125:     loadw  SVAD_GCLK,r1     # R1 = system time
2126:     loadq  $PARM_LONG,r2    # R2 = message parameter is a long
2127:     b      Crash            # Crash the system
2128:
2129: _pfmsg:
2130:     .byte  NL,NL,'*,' ,'P,'o,'w,'e,'r,' ,'F,'a,'i,'l,'u,'r,'e,' ,'-
2131:     .byte  ' ,'T,'i,'m,'e,' ,'=,' ,0
```

9.4.14.3 I_Clock - Clock Interrupt Handler

A clock interrupt is received **CLK_FREQ** times every second. **I_Clock** simply increments the system clock in the global data page (at **SVAD_GCLK**) and the user running time (at **U_RunTime**) and returns. It only saves the **r0** register, since that is all it uses.

```
2132: #
2133: # System Clock Interrupt.  Taken CLK_FREQ times per second.
2134: #
2135:     .align 2
2136: I_Clock:
2137:     pushw  r0,sp            # Save our scratch register
2138:     loadw  SVAD_GCLK,r0     # Increment the system clock
2139:     addq   $1,r0
2140:     storw  r0,SVAD_GCLK
2141:     loadw  U_RunTime,r0     # Increment the program run time
2142:     addq   $1,r0
2143:     storw  r0,U_RunTime
2144:     popw   sp,r0            # Restore user's register
2145:     reti   sp               #  and exit
```

9.4.15 Exception Handler Support Routines

This section contains support routines that the exception handlers use to perform their functions. Included here are routines that handle multiple traps and fatal errors detected or caused by exception handlers. Also included are routines that retrieve data from the user virtual address space.

9.4.15.1 ChkMTrap - Check for and Process Data Memory Traps

The floating-point trap handlers call **ChkMTrap** to check for and process simultaneous data memory traps. **ChkMTrap** retrieves the **PSW_MTS** field from the **PSW** register in the INTRAP stack frame. If this field is non-zero, it indicates the presence of a data memory trap hidden behind the floating-point error that was signalled. If the field is zero, there is not a multiple trap condition, so **ChkMTrap** returns.

To process the simultaneous data memory trap, **ChkMTrap** converts the **PSW_MTS** field into a table offset and calls the appropriate error process through the **_dmtproc** table. The instruction at line 2165 converts the **PSW_MTS** field into a table offset. The operation is clearer if you think of it in two steps:

1. Shift the Memory Trap Status field right by **SHF_PSW_MTS** to convert it into an index.
2. Shift the index left by two bits (that is, multiply it by four) to convert it into a Data Memory Trap Processor table offset.

If the data memory trap processor returns, **ChkMTrap** returns to the floating-point trap handler.

```
2146: # ********************************************
2147: # *                                          *
2148: # *   Exception Handler Auxiliary Routines.  *
2149: # *                                          *
2150: # ********************************************
2151:
2152: #
2153: # Check for and process data memory trap.  Called from Floating Point trap
2154: # handlers to process multiple traps.
2155: # Expects:
2156: #   R14 = address of INTRAP stack frame
2157: # Returns:
2158: #   Nothing
2159: #
2160:     .align 2
2161: ChkMTrap:
2162:     loadw  INTRAP_PSW(r14),r0 # R0 = stacked PSW
2163:     andi   $PSW_MTS,r0        # R0 = memory trap status
2164:     breq   _cmtdone           # There wasn't a memory trap
2165:     shli   $(-SHF_PSW_MTS+2),r0 # R0 = offset into jump table
2166:     call   sp,DMTProc(r0)     # Execute data memory trap processor
2167: _cmtdone:
2168:     ret    sp                 # Return to Floating Point trap processor
2169:
2170:     .align 2
2171: DMTProc:
2172:     .long  InvMTS             # PSW_MTS_00;     Invalid MTS value
2173:     .long  CMemErr            # PSW_MTS_CME;    Corrected Memory Error
2174:     .long  UMemErr            # PSW_MTS_UCME;   Uncorrectable Memory Error
2175:     .long  InvMTS             # PSW_MTS_03;     Invalid MTS value
2176:     .long  InvMTS             # PSW_MTS_04;     Invalid MTS value
2177:     .long  DPageFlt           # PSW_MTS_PGFLT;  Page Fault (Data Mem)
2178:     .long  DRdProt            # PSW_MTS_RDFLT;  Read Protect Fault (Data Mem)
2179:     .long  DWrtProt           # PSW_MTS_WRFLT;  Write Protect Fault (Data Mem)
2180:     .long  InvMTS             # PSW_MTS_08;     Invalid MTS value
2181:     .long  InvMTS             # PSW_MTS_09;     Invalid MTS value
2182:     .long  InvMTS             # PSW_MTS_0A;     Invalid MTS value
2183:     .long  InvMTS             # PSW_MTS_0B;     Invalid MTS value
2184:     .long  InvMTS             # PSW_MTS_0C;     Invalid MTS value
2185:     .long  InvMTS             # PSW_MTS_0D;     Invalid MTS value
2186:     .long  InvMTS             # PSW_MTS_0E;     Invalid MTS value
2187:     .long  InvMTS             # PSW_MTS_0F;     Invalid MTS value
```

9.4.15.2 InvMTS - Process an Invalid MTS Condition

ChkMTrap calls **InvMTS** indirectly through the **_dmtproc** table if the value in the **PSW_MTS** field is invalid during multiple exception processing. **InvMTS** simply prints the message containing the invalid Memory Trap Status value and crashes the system.

9.4.15.3 ExFatal - Process Fatal Exception

Various exception handlers call **ExFatal** to differentiate between fatal user and Monitor errors. **ExFatal** simply checks the state of the previous bit (**SSW_P**) in the supervisor **SSW** register. If the bit is set, it indicates that the fatal error was caused by the user program. In this case, **ExFatal** branches to **ExAbort** to terminate the program. If the bit is clear, it indicates that the fatal error was caused by the Monitor. In this case, **ExFatal** branches to **ExCrash**, to crash the system.

```
2188: #
2189: # Process an invalid MTS condition.
2190: #
2191:     .align 2
2192: InvMTS:
2193:     loada  _imtsmsg,r0      # R0 = message address
2194:     loadw  INTRAP_PSW(r14),r1
2195:     andi   $PSW_MTS,r1      # R1 = MTS value
2196:     shli   $-SHF_PSW_MTS,r1 # Put into low order bits
2197:     loadq  $PARM_BYTE,r2    # Parameter is 8 bits long
2198:     b      ExCrash          # Crash the system
2199:
2200: _imtsmsg:
2201:     .byte  '*,' ,'I,'n,'v,'a,'l,'i,'d,' ,'M,'e,'m,'o,'r,'y
2202:     .byte  ' ,'T,'r,'a,'p,' ,'S,'t,'a,'t,'u,'s,' ,'=,' ,0
2203:
2204: #
2205: # Process a fatal error (from an exception handler).
2206: #
2207: # Expects:
2208: #   R0 = Address of message.
2209: #   R1 = Parameter (if present)
2210: #   R2 = Parameter flag
2211: # Returns:
2212: #   Does not return
2213: #
2214:     .align 2
2215: ExFatal:
2216:     movpw  ssw,r3
2217:     andi   $SSW_P,r3        # Was trap taken in User mode?
2218:     brne   ExAbort          #  Yes - abort the user program
2219:     b      ExCrash          #  No - crash the system
```

9.4.15.4 ExCrash - Crash System from Exception Handler

Various exception handlers call **ExCrash** to crash the system in the event of a fatal Monitor error. **ExCrash** saves its entry parameters and calls **DumpEStk** to dump the contents of the exception (INTRAP) stack frame. Then, it restores its parameters and branches unconditionally to **Crash**, which displayes the diagnostic message and crashes the system.

9.4.15.5 ExAbort - Abort User Program from Exception Handler

Various exception processors call **ExAbort** to terminate the user program in the event of a fatal user error. **ExAbort** saves its entry parameters and calls **DumpEStk** to dump the contents of the exception (INTRAP) stack frame. Then, it restores its parameters and branches unconditionally to **Abort**, which displayes the diagnostic message and terminates the user program.

```
2220: #
2221: # Crash the system from an exception handler.
2222: # Expects:
2223: #   R0 = Address of message.
2224: #   R1 = Parameter (if present)
2225: #   R2 = Parameter flag
2226: #   R14 = Address of trap stack frame
2227: # Returns:
2228: #   Does not return
2229: #
2230:     .align 2
2231: ExCrash:
2232:     pushw  r0,sp
2233:     pushw  r1,sp
2234:     pushw  r2,sp
2235:     call   sp,DumpEStk      # Dump Trap stack frame
2236:     popw   sp,r2
2237:     popw   sp,r1
2238:     popw   sp,r0
2239:     b      Crash            # Crash the system
2240:
2241: #
2242: # Abort the user program from an exception handler.
2243: # Expects:
2244: #   R0 = Address of message.
2245: #   R1 = Parameter (if present)
2246: #   R2 = Parameter flag
2247: #   R14 = Address of trap stack frame
2248: # Returns:
2249: #   Does not return
2250: #
2251:     .align 2
2252: ExAbort:
2253:     pushw  r0,sp
2254:     pushw  r1,sp
2255:     pushw  r2,sp
2256:     call   sp,DumpEStk      # Dump Trap stack frame
2257:     popw   sp,r2
2258:     popw   sp,r1
2259:     popw   sp,r0
2260:     b      Abort            # Abort the user program
```

9.4.15.6 GetUserB - Retrieve Byte from User Virtual Memory

GetUserB retrieves 8 bits of data from the user's virtual address space. The routine expects the user virtual address in register **r0**. It gets the current **SSW**, sets the **SSW_UU** bit (which allows the Monitor to access user virtual memory), retrieves an unsigned byte from user virtual memory into register **r0**, turns off the **SSW_UU** bit, and returns. Remember that you cannot access supervisor virtual memory as long as the **SSW_UU** bit is set. It is best to set it, make all necessary references to user memory, and then clear it again immediately.

9.4.15.7 GetUserH - Retrieve Halfword from User Virtual Memory

GetUserH retrieves 16 bits of data from the user's virtual address space. The routine expects the user virtual address in register **r0**. It sets the **SSW_UU** bit (which allows the Monitor to access user virtual memory), retrieves an unsigned halfword from user virtual memory into register **r0**, turns off the **SSW_UU** bit, and returns. Remember that you cannot access supervisor virtual memory as long as the **SSW_UU** bit is set. It is best to set it, make all necessary references to user memory, and then clear it again immediately.

```
2261: #
2262: # Get one byte from user space.
2263: # Expects:
2264: #   R0 = user virtual address
2265: # Returns:
2266: #   R0 = byte from user space
2267: #
2268:     .align 2
2269: GetUserB:
2270:     movpw  ssw,r1
2271:     ori    $SSW_UU,r1
2272:     movwp  r1,ssw           # Turn on the UU bit
2273:     loadbu (r0),r0          # R0 = byte from user space
2274:     andi   $~SSW_UU,r1
2275:     movwp  r1,ssw           # Turn off the UU bit
2276:     ret    sp
2277:
2278: #
2279: # Get one halfword from user space.
2280: # Expects:
2281: #   R0 = user virtual address
2282: # Returns:
2283: #   R0 = halfword from user space
2284: #
2285:     .align 2
2286: GetUserH:
2287:     movpw  ssw,r1
2288:     ori    $SSW_UU,r1
2289:     movwp  r1,ssw           # Turn on the UU bit
2290:     loadhu (r0),r0          # R0 = halfword from user space
2291:     andi   $~SSW_UU,r1
2292:     movwp  r1,ssw           # Turn off the UU bit
2293:     ret    sp
```

9.4.16 Monitor Data Segment

The monitor data segment contains all of the writable monitor variables. Some high-level language compilers do not differentiate between read-only and read-write data. Instead, they combine all data together in the data segment. This example allows read-only data (primarily initialization tables and error messages) in the text (code) segment. Only variables which must be written are placed into the data segment. This ensures that constants cannot be altered inadvertently, since text pages allow read/execute permission only. This does allow a runaway program to attempt to execute data, however. Perhaps the ideal solution would be to place only executable instructions in the text segment, read/write variables in the data segment, and read-only constants in a separate, read-only page.

9.4.16.1 Main Memory Usage Tables

These two tables control the allocation, sharing, and deallocation of real memory pages. Each table is an array, indexed by real page number, with a single byte reserved for each page of real memory in the system.

9.4.16.2 I/O Space Usage Tables

These two tables control the allocation, sharing, and deallocation of I/O Space pages. Each table is an array, indexed by I/O Space page number, with a single byte reserved for each page of I/O Space in the system.

9.4.16.3 Boot Space Usage Tables

These two tables control the allocation, sharing, and deallocation of Boot Space pages. Each table is an array, indexed by Boot Space page number, with a single byte reserved for each page of Boot Space in the system.

9.4.16.4 Miscellaneous Monitor Variables

The Monitor stores the contents of the **SP** register in **CSwitchSP**, immediately before performing a context switch to the user program. The user program termination handler (either in **SC_Exit** or in **Abort**) restores the Monitor stack pointer from this variable and executes a **reti** instruction, returning control to the **CSwitch** routine in supervisor mode.

The floating-point trap handlers retrieve the **PC** of the faulting instruction and store it in **FPTrapPC** immediately before calling **ChkMTrap** to check for and process simultaneous data memory faults.

```
2294: # ****************************************
2295: # *                                      *
2296: # *   Monitor Data Segment Begins Here.  *
2297: # *                                      *
2298: # ****************************************
2299:     .data
2300: Data:
2301: #
2302: # Real Memory Page Usage and Count Tables.
2303: #
2304: RMUsage:
2305:     .space RM_PAGES         # One byte for each page of real memory
2306: RMUseCnt:
2307:     .space RM_PAGES
2308:
2309: #
2310: # I/O Space Page Usage and Count Tables.
2311: #
2312: IOUsage:
2313:     .space IO_PAGES         # One byte for each page of I/O Space
2314: IOUseCnt:
2315:     .space IO_PAGES
2316:
2317: #
2318: # Bootstrap Space Page Usage and Count Tables.
2319: #
2320: BSUsage:
2321:     .space BS_PAGES         # One byte for each page of Bootstrap Space
2322: BSUseCnt:
2323:     .space BS_PAGES
2324:
2325: # Value of SP before context switch
2326:     .align 4
2327: CSwitchSP:
2328:     .long  0
2329:
2330: # PC for Floating Point Traps.
2331: FPTrapPC:
2332:     .long  0
```

9.4.16.5 User Program Information

The Monitor stores information about the user program in these variables.

U_RunTime is the total number of clock ticks that have occurred since the user was given the CPU.

U_DataLen is the total length (in pages) of the user's data segment. This number will grow by one for each successful **SYS_EXDS** system call executed on behalf of the user.

U_StkLen is the total length (in pages) of the user's stack segment. This number will grow by one each time the data page fault trap handler extends the user's stack.

U_TDataPg is the user virtual page number of the highest page in the user's data segment. This number will grow by one for each successful **SYS_EXDS** system call executed on behalf of the user.

U_BStkPg is the user virtual page number of the lowest page in the user's stack segment. This number will be reduced by one each time the data page fault trap handler extends the user's stack.

```
2333: # *******************************************
2334: # *                                         *
2335: # *   Information about the user process.   *
2336: # *                                         *
2337: # *******************************************
2338:     .align 4
2339: U_RunTime:
2340:     .long  0                # Number of seconds the user has been running
2341: U_DataLen:
2342:     .word  0                # User's data segment length (pages)
2343: U_StkLen:
2344:     .word  0                # User's stack segment length (pages)
2345: U_TDataPg:
2346:     .word  0                # Virtual page # of top of user's data seg
2347: U_BStkPg:
2348:     .word  0                # Virtual page # of bottom of user's stack seg
```

9.5 AN EXAMPLE USER PROGRAM

This section contains the annotated listing of a simple user program. The program compares the execution times of two different algorithms for calculating a bounded sequence of Fibonacci numbers. (See Donald Knuth's *The Art of Computer Programming*, *Vol. 1,* Section 1.2.8 for a discussion of Fibonacci numbers.)

In the sequence in Table 9.1, each number is the sum of the preceding two numbers. The sequence is represented by F_n, which is defined formally in Table 9.2. Since the sequence is defined recursively (each Fibonacci number is described in terms of previous Fibonacci numbers), it naturally suggests a recursive solution. This is an elegant approach. In a language such as LISP, it could be implemented by a single statement. But each recursion requires a function call with a certain amount of stack manipulation to preserve its caller's registers. Such a solution is inefficient in both time and space.

The Fibonacci sequence can also be calculated iteratively if the program stores the two most recent results. This approach is more efficient, requiring no function calls and no stack space.

The example user program calculates the first **MAXFIB** numbers in the Fibonacci sequence using both an iterative algorithm and a recursive one. Although the Fibonacci sequence is defined for integers, the program uses double precision floating-point arithmetic since the numbers are large. Also, both algorithms store the generated sequence in an array.

TABLE 9.1 Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

TABLE 9.2 Fibonacci Formal Definition

$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2},\ n >= 2.$

9.5.1 Memory Usage

The user program allocates an extra page of data with the **SYS_EXDS** system call. In addition, if **MAXFIB** is large enough, the recursive algorithm will cause a data page fault in its stack segment.

Figure 9.17 illustrates the final user virtual memory usage assuming that there has been one **SYS_EXDS** call and one stack page fault. Figure 9.18 illustrates the user's virtual memory map under the same conditions.

Figure 9.18 Final User Virtual Memory Map

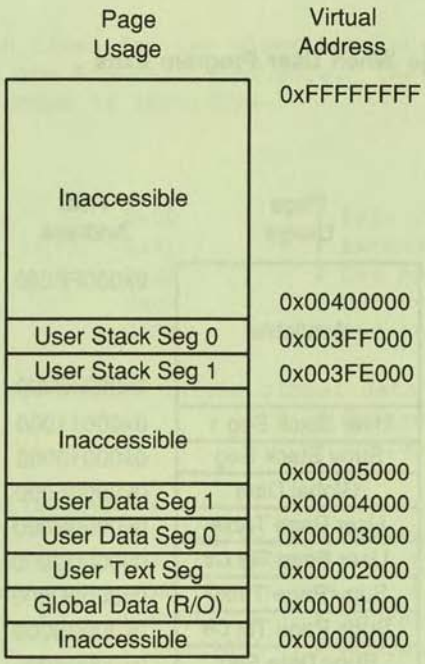


Figure 9.17 Final User Virtual Memory Usage

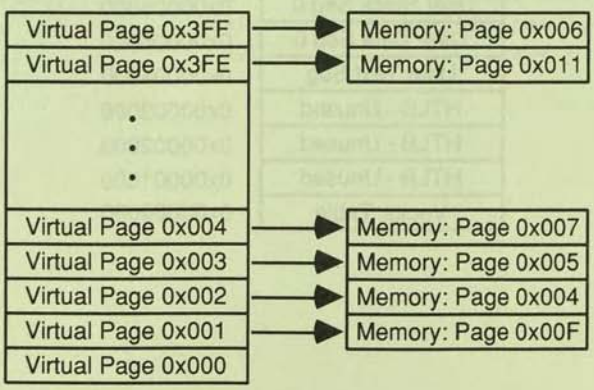
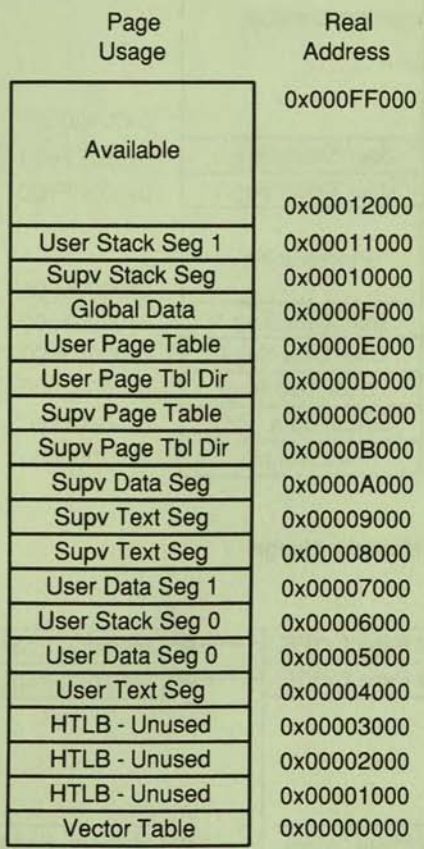


Figure 9.19 illustrates the real memory usage when the user program exits and before the Monitor has freed the user memory.

9.5.2 Prologue

The code on the facing page defines the constants used by the Fibonacci program. The **SYS_XXXX** and **UVAD_XXXX** definitions are identical to those in the Monitor. **MAXFIB** is the length of the Fibonacci sequence to generate. **BUFLEN** is the length of the decimal ASCII conversion buffer.

Figure 9.19 Real Memory Usage When User Program Exits



```
0001: # **********************************
0002: # *                                *
0003: # *   Example User Mode Program.   *
0004: # *                                *
0005: # **********************************
0006:
0007: #
0008: # fib.s
0009: #
0010: # Compares execution times for two algorithms that generate the first
0011: # MAXFIB numbers in the Fibonacci sequence.  The first algorithm is
0012: # sequential; the second is recursive.
0013: #
0014:
0015: # System Calls
0016:     .set   SYS_EXIT,        0x00          # Exit from user program
0017:     .set   SYS_EXDS,        0x01          # Extend data segment
0018:     .set   SYS_GETB,        0x02          # Get byte from SysIn
0019:     .set   SYS_PUTB,        0x03          # Put byte to SysOut
0020:     .set   SYS_PUTS,        0x04          # Put string to SysOut
0021:
0022: # Fixed offsets to the values in the global data page.
0023:     .set   G_CLK,           0             # System clock
0024:     .set   G_NAM,           4             # System name
0025:
0026: # Fixed user virtual addresses.
0027:     .set   UVAD_GDAT,       0x00001000    # Address of global data
0028:     .set   UVAD_GCLK,       UVAD_GDAT+G_CLK     # Global system clock
0029:     .set   UVAD_GNAM,       UVAD_GDAT+G_NAM     # Global system name
0030:     .set   UVAD_TEXT,       0x00002000    # Address of text segment
0031:     .set   UVAD_DATA,       0x00003000    # Address of data segment
0032:     .set   UVAD_STK,        0x003FF000    # Top of user stack
0033:
0034: # Program constants.
0035:     .set   MAXFIB,          25            # Lenghth of sequence to calculate
0036:     .set   BUFLEN,          16            # Decimal ASCII conversion buffer
0037:     .set   NL,              0x0A          # ASCII newline character
0038:     .set   SP,              0x20          # ASCII space character
```

9.5.3 The Main Body

The user program:

- receives control from the Monitor at address **Start** and calls **SignOn** to print a sign-on message.
- issues a **SYS_EXDS** system call to acquire a 4,096 byte page of data space.
- times the execution of the iterative algorithm (**CalcFibI**) and displays the result.
- times the execution of the recursive algorithm (**CalcFibR**) and displays the result.
- issues a **SYS_EXIT** system call to return control to the Monitor.

```
0039:     .text
0040:     .globl Start
0041:     .align 2
0042: Start:
0043:     call   sp,SignOn        # Print the signon information
0044:
0045:     calls  $SYS_EXDS        # Extend the data segment with another page
0046:     cmpq   $0,r0
0047:     bceq   _nopage          # Couldn't allocate memory for the array
0048:     storw  r0,FNumber       # Save array address
0049:
0050:     loadi  $PSW_FPE,r0
0051:     movwp  r0,psw           # Enable all floating-point traps
0052:
0053:     loada  _fibimsg,r0
0054:     calls  $SYS_PUTS
0055:     loadw  FNumber,r0       # Array address
0056:     loadw  UVAD_GCLK,r1
0057:     storw  r1,ClockStart    # Save the start time
0058:     call   sp,CalcFibI      # Get the Fibonacci sequence non-recursively
0059:     loadw  UVAD_GCLK,r1
0060:     storw  r1,ClockStop     # Save the stop time
0061:     call   sp,PrintET       # Print the elapsed time for Algorithm 1
0062:
0063:     loada  _fibrmsg,r0
0064:     calls  $SYS_PUTS
0065:     loadw  FNumber,r0       # Array address
0066:     loadw  UVAD_GCLK,r1
0067:     storw  r1,ClockStart    # Save the start time
0068:     call   sp,CalcFibR      # Get the Fibonacci sequence recursively
0069:     loadw  UVAD_GCLK,r1
0070:     storw  r1,ClockStop     # Save the stop time
0071:     call   sp,PrintET       # Print the elapsed time for Algorithm 2
0072:
0073:     loadq  $0,r0            # Exit status code
0074: _exit:
0075:     calls  $SYS_EXIT
```

9.5.4 SignOn - Print a sign-on message

SignOn issues a **SYS_PUTS** call to print a string on the **SysOut** device. It then calls **ConvertDA** to convert **MAXFIB** into an ASCII string and issues another **SYS_PUTS** call to print it.

```
0076: _nopage:
0077:     loada  _nopmsg,r0
0078:     calls  $SYS_PUTS
0079:     loadq  $1,r0              # Exit status code
0080:     b      _exit
0081:
0082: _nopmsg:
0083:     .byte  'U,'n,'a,'b,'l,'e,' ,'t,'o,' ,'e,'x,'t,'e,'n,'d
0084:     .byte  ' ,'d,'a,'t,'a,' ,'s,'e,'g,'m,'e,'n,'t,'.,NL,0
0085:
0086: _fibimsg:
0087:     .byte  'I,'t,'e,'r,'a,'t,'i,'v,'e
0088:     .byte  ' ,'A,'l,'g,'o,'r,'i,'t,'h,'m,' ,'-,SP,0
0089:
0090: _fibrmsg:
0091:     .byte  'R,'e,'c,'u,'r,'s,'i,'v,'e
0092:     .byte  ' ,'A,'l,'g,'o,'r,'i,'t,'h,'m,' ,'-,SP,0
0093:
0094: #
0095: # Print the sign-on message.
0096: #
0097:     .align 2
0098: SignOn:
0099:     loada  _somsg,r0          # Print the signon message
0100:     calls  $SYS_PUTS
0101:     loadi  $MAXFIB,r0
0102:     call   sp,ConvertDA       # Convert decimal to ASCII, R0=addr of string
0103:     calls  $SYS_PUTS          # Print number
0104:     loadi  $NL,r0
0105:     calls  $SYS_PUTB          # Print new line
0106:     ret    sp
0107:
0108: _somsg:
0109:     .byte  'F,'i,'b,'o,'n,'a,'c,'c,'i,' ,'S,'e,'q,'u,'e,'n,'c,'e
0110:     .byte  ' ,'G,'e,'n,'e,'r,'a,'t,'o,'r,'.,' ,' ,'L,'i,'m,'i,'t
0111:     .byte  ' ,'=,' ,0
```

9.5.5 CalcFibI - Calculate Fibonacci Sequence Iteratively

CalcFibI calculates the first **MAXFIB** Fibonacci numbers using an iterative algorithm. It implements the C language function in Example 9.14.

The loop iterates from two to **MAXFIB** minus one, adding up the two previous values and storing the result into the position for the current value.

An iterative solution does not depend upon storing all *N* Fibonacci numbers: it need only store the previous two. This implementation stores all generated values for simplicity.

Example 9.14

```
#define   FPZero            0.0
#define   FPOne             1.0

void FibI(FNumber)
double *FNumber;
{
     int i;

     FNumber[0]= FPZero;
     FNumber[1]= FPOne;
     for (i=MAXFIB-2; i>0; i--,FNumber++)
          *FNumber = FNumber[-1] + FNumber[-2];
}
```

```
0112: #
0113: # Calculate the Fibonacci sequence iteratively.
0114: # Expects:
0115: #    R0 = base of storage array
0116: # Returns:
0117: #    Nothing
0118: #
0119:      .align 2
0120: CalcFibI:
0121:      loadd  DPZero,f0
0122:      stord  f0,(r0)          # F[0]
0123:      loadd  DPOne,f0
0124:      stord  f0,8(r0)         # F[1]
0125:      addi   $16,r0           # R0 is address of F[n]
0126:      loadi  $MAXFIB-2,r1     # R1 is loop count control
0127: _cfiloop:
0128:      loadd  -8(r0),f0        # F0 = F[n-1]
0129:      loadd  -16(r0),f1       # F1 = F[n-2]
0130:      addd   f1,f0            # F0 = F[n]
0131:      stord  f0,(r0)          # Save F[n]
0132:      addq   $8,r0
0133:      subq   $1,r1
0134:      brgt   _cfiloop
0135:      ret    sp
0136:
```

9.5.6 CalcFibR - Calculate Fibonacci Sequence Recursively

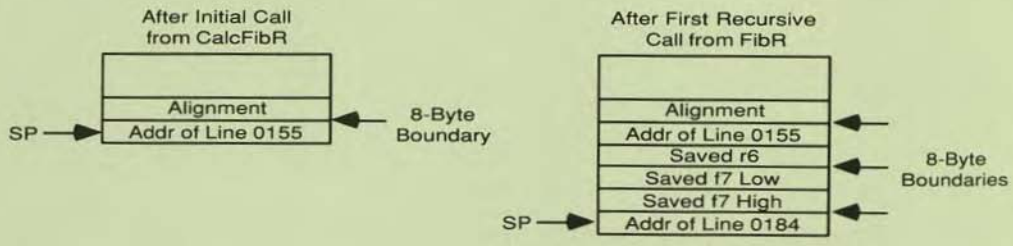
CalcFibR calculates the first **MAXFIB** Fibonacci numbers using a recursive algorithm. It implements the C language function in Example 9.15.

The loop iterates from zero to **MAXFIB** minus one, calling **FibR** to calculate the current Fibonacci number and storing it into the position for the current value.

The CLIPPER architecture requires that all data objects be aligned on memory addresses that are evenly divisible by the size of the object. Thus, 32-bit words must be stored on a 4-byte boundary, and double precision floating-point numbers must be stored on an 8-byte boundary. Because of this requirement, the stack must be aligned properly before a program executes the **saved*n*** instruction. **CalcFibR** knows that the **FibR** routine requires one floating-point register for storage, so it aligns the stack *before* it enters its inner loop. **FibR** maintains the stack alignment throughout its recursive calls to itself, so the **saved7** / **restd7** instruction pair always produces the correct results. Figure 9.20 illustrates the stack upon entry to the **FibR** routine.

The alignment operation is performed at lines 147 through 149. Depending upon the stack alignment in **CalcFibR**, the value in **r1** after executing the **andw** at line 148 will be either zero or four. Thus, the **subw** at line 149 either does nothing or decrements the **SP** by four. The latter condition is illustrated in Figure 9.20. Since the decrement operation always results in an 8-byte alignment of the stack, and since no more that one long word can ever be wasted, the **call** instruction at line 154 never results in a data page fault. In general, however, reserving stack space can cause a page fault on the next stack reference. When the amount of reserved space is larger than 4,096 bytes, the data page fault handler must be capable of extending the stack segment by more than one page in order to resolve the fault. The example Monitor program does not perform multiple page allocations on stack faults. If it cannot resolve a stack fault by allocating one page to the user's stack segment, the Monitor simply aborts the offending program.

Figure 9.20 Stack Upon Entry to FibR



Example 9.15

```
CalcFibR(FNumber)
double *FNumber;
{
     double FibR();
     int n;

     for (n=0; n<MAXFIB; n++)
          *FNumber++ = FibR(n);
 }
```

```
0137: #
0138: # Calculate the Fibonacci sequence recursively.
0139: # Expects:
0140: #    R0 = address of array to store results
0141: # Returns:
0142: #    Nothing
0143: #
0144:      .align 4
0145: CalcFibR:
0146:      movw   r0,r6            # Save array address
0147:      loadq  $7,r1
0148:      andw   sp,r1
0149:      subw   r1,sp            # Align SP to 8-byte boundary
0150:      storw  r1,StackAl       # Save the alignment value
0151:      loadq  $0,r7
0152: _cfrloop:
0153:      movw   r7,r0            # R0 = n
0154:      call   sp,FibR          # F0 = F[n] (Stack on a 4-byte boundary)
0155:      stord  f0,(r6)          # Save it
0156:      addq   $8,r6            # Increment the array address
0157:      addq   $1,r7            #  and n
0158:      cmpi   $MAXFIB,r7
0159:      bcgt   _cfrloop
0160:      loadw  StackAl,r0       # R0 = stack alignment value
0161:      addw   r0,sp            # Restore the SP
0162:      ret    sp
```

9.5.7 FibR - Calculate Fibonacci Number Recursively

FibR calculates F_n by recursively calling itself to calculate F_{n-1} and F_{n-2}, and then summing the results. It implements the C language function in Example 9.16.

If *N* is less than 2, **Fib** returns *N* converted to double precision. Otherwise, **Fib** saves its working registers and then calls itself twice; once with *N* - 1 as a parameter, and again, with *N* - 2. Each recursive call to **Fib** consumes 16 bytes of stack space as follows:

- 4 bytes for the return address.
- 4 bytes for temporary register **r6**.
- 8 bytes for temporary floating-point register **f7**.

Example 9.16

```
double
FibR(n)
int n;
{
     switch (n) {
     case 0:
     case 1:
          return ((double)n);
          /* NOTREACHED */
     default:
          return (FibR(n-1) + FibR(n-2));
          /* NOTREACHED */
     }
}
```

```
0163: #
0164: # Recursively Calculate the Fibonacci number of n.
0165: # Expects:
0166: #   R0 = n
0167: #   SP on a 4-byte boundary
0168: # Returns:
0169: #   F0 = F[n]
0170: #
0171:     .align 2
0172: FibR:
0173:     cmpq   $2,r0            # n < 2?
0174:     bcle   __recurse        #  No - calculate recursively
0175:     cnvwd  r0,f0            #  Yes - F[n] = (double)n
0176:     ret    sp
0177:
0178: __recurse:
0179:     pushw  r6,sp            # Get a general register
0180:     saved7                  # Get a floating point register
0181:     movw   r0,r6            # Save n
0182:     subq   $1,r0            # R0 = n-1
0183:     call   sp,FibR          # F0 = F[n-1]
0184:     movd   f0,f7            # Save it
0185:     movw   r6,r0            # R0 = n
0186:     subq   $2,r0            # R0 = n-2
0187:     call   sp,FibR          # F0 = F[n-2]
0188:     addd   f7,f0            # F0 = F[n-1] + F[n-2] = F[n]
0189:     restd7                  # Restore F7
0190:     popw   sp,r6            # Restore R6
0191:     ret    sp
```

9.5.8 PrintET - Print Elapsed Time Message.

PrintET prints the **_etmsg** string, calculates the elapsed time and converts it into an ASCII string, outputs the string, and terminates the message with a new line character.

```
0192: #
0193: # Print the elapsed time for the execution of an algorithm
0194: # Expects:
0195: #   ClockStart - set at beginning of algorithm
0196: #   ClockStop - set at end of algorithm
0197: # Returns:
0198: #   Nothing
0199: #
0200:     .align 2
0201: PrintET:
0202:     loada  _etmsg,r0
0203:     calls  $SYS_PUTS
0204:     loadw  ClockStop,r0      # R0 = Stop time
0205:     loadw  ClockStart,r1     # R1 = Start time
0206:     subw   r1,r0             # R0 = elapsed time for algorithm
0207:     call   sp,ConvertDA      # R0 = address of ASCII string
0208:     calls  $SYS_PUTS
0209:     loadi  $NL,r0
0210:     calls  $SYS_PUTB         # New line
0211:     ret    sp
0212:
0213: _etmsg:
0214:     .byte  'E,'l,'a,'p,'s,'e,'d,' ,'T,'i,'m,'e,' ,'=,' ,0
```

9.5.9 ConvertDA - Convert Value to Decimal ASCII

ConvertDA converts a 32-bit value into a null-terminated ASCII string. It starts by loading the address of the high end of a convertsion buffer (**DABuffer+DABUFLEN-1**) into register **r1**, storing the null terminator at that address, and then loading the constant 10 (the number base) into register **r2**.

ConvertDA then loops until the value equals zero. For each iteration, the routine:

1. Decrements the string pointer.
2. Takes the modulus of the remaining value and the base.
3. Converts the result of step 2 into ASCII by adding the representation of the number zero.
4. Stores the new ASCII number into the buffer indirectly through the pointer.
5. Reduces the remaining value by dividing it by the base.
6. If the result of step 5 is not equal to zero, the routine loops.
7. When the value goes to zero, **ConvertDA** returns the value of the base of the string in register **r0**.

```
0215: #
0216: # Convert decimal value into ASCII.
0217: # Expects:
0218: #   R0 = decimal value to convert
0219: # Returns:
0220: #   R0 = address of null-terminated ASCII result string
0221: #
0222: #
0223:     .align 2
0224: ConvertDA:
0225:     loada  (Buffer+BUFLEN-1),r1    # R1 = top end of result buffer
0226:     loadq  $0,r2
0227:     storb  r2,(r1)           # String terminator
0228:     loadq  $10,r2            # R2 = base
0229: _cdaloop:
0230:     subq   $1,r1             # Decrement the string pointer
0231:     movw   r0,r3
0232:     modwu  r2,r3             # R3 = next digit
0233:     addi   $'0,r3            # Convert to ASCII digit
0234:     storb  r3,(r1)           # Store it
0235:     divwu  r2,r0             # Reduce the value
0236:     cmpq   $0,r0
0237:     bcne   _cdaloop
0238:     movw   r1,r0             # R0 = address of result
0239:     ret    sp
```

9.5.10 User Data Segment

The **ClockStart** and **ClockStop** variables record the starting and ending times of the algorithm under test. Remember that the Monitor keeps the time in ticks, not seconds.

ConvertDA uses the **DABuffer** to store its null-terminated decimal ASCII string. Since there is only one buffer, and its address is coded into **ConvertDA**, each conversion destroys the results of the previous one.

FPZero and **FPOne** hold the double precision constants 0.0 and 1.0 respectively.

```
0240: # ************************************
0241: # *                                  *
0242: # *   User Data Segment Starts Here. *
0243: # *                                  *
0244: # ************************************
0245:     .data
0246:     .align 4
0247: # Based address of storage array.
0248: FNumber:
0249:     .long  0
0250:
0251: # Algorithm start time.
0252: ClockStart:
0253:     .long  0
0254:
0255: # Algorithm stop time.
0256: ClockStop:
0257:     .long  0
0258:
0259: # Stack alignment value.
0260: StackAl:
0261:     .long  0
0262:
0263: Buffer:
0264:     .space BUFLEN
0265:
0266: # Double precision constants as variables.
0267:     .align 8
0268: DPZero:
0269:     .double0d0.0
0270:
0271: DPOne:
0272:     .double0d1.0
```

9.6 CROSS REFERENCE LISTINGS

This section contains cross reference listings for both the Monitor and the user program. Each cross reference contains a sorted list of all names used in the program. Following each name is a sorted list of line numbers on which the name occurs. The line on which the name is defined is enclosed in parentheses.

9.6.1 Monitor Cross Reference

```
- A -
Abort          (1381) 2260
_abtmsg        1383   (1388)
AllocPage      0605   (0861) 1239   1278   1657   2012
_apbaduc       0878   (0887)
_apbmsg        0889   (0893)
_apdone        0873   (0883)
_apfound       0868   (0875)
_aploop        (0865) 0871

- B -
BS_PAGES       (0025) 0175   0189   1094   2321   2323
BSUsage        0176   1092   (2320)
BSUseCnt       0190   1093   (2322)

- C -
C_D_RS         0246   0446
C_D_SPDO       0216
C_D_UPDO       0233
C_D_VFA        1628   1709   1734
ChkMTrap       1757   1777   1795   1815   1833   (2161)
C_I_RS         0247   0447
C_I_SPDO       0215
C_I_UPDO       0232
C_I_VFA        1864   1881
CLK_FREQ       (0022)
ClrWindow      1285   (1299)
CMD_HELP       (0154)
CMD_QUIT       (0153)
CMemErr        1566   (1577)
_cmemsg        1578   (1584)
_cmtdone       2164   (2167)
CR             (0144)
Crash          0563   0847   0891   0950   1005   1174   (1402) 2127   2239
_crmsg         1404   (1415)
C_RS_ALL       0245
C_RS_RUV       0445
CSwExit        0428   (0459)
CSwitch        0342   (0426)
CSwitchSP      0434   1385   1993   (2327)
```

- D -

Data (2300)
_dePCmsg 1456 (1473)
_dePSWmsg 1466 (1477)
_deSSWmsg 1461 (1475)
DMTProc (2171)
DPageFlt 1615 (1627)
_dpfabort 1643 1646 (1681)
_dpfbmsg 1678 (1695)
_dpfexit 1675 1679 (1683)
_dpfmsg 1635 1682 (1687)
_dpfnadj 1651 (1655)
_dpfnopg 1658 (1673)
_dpfsmsg 1674 (1691)
_dpfstkb 1663 (1677)
_dpfuser 1631 1634 (1639)
DRdProt (1707)
DumpEStk (1455) 1947 2235 2256
DWrtProt (1732)

- E -

_erunmsg 0344 (0354)
ExAbort 1685 1713 1716 1738 1741 1932 2218 (2252)
ExCrash 1552 1602 1637 1717 1742 1974 2112 2198 2219 (2231)
ExecUser (0335)
_exfail 0337 (0357)
ExFatal 1761 1781 1799 1819 1837 1852 1866 1883 1911 (2215)
_exfmsg 0358 (0362)
_exmsg 1989 (1996)
_exprmsg 1880 (1885)
_extdone (2028) 2036
_extfail 2013 2018 (2034)

- F -

_fpbadp 0922 (0937)
_fpbaduc 0929 (0941)
_fpbpmsg 0938 (0952)
_fpbucmsg 0942 (0956)
_fpd0msg (1801)
_fpdie 0939 0943 (0946)
_fpdone 0909 0913 (0934)
_fpfmsg 0945 (0960)
_fpfree 0926 (0944)
_fpimsg 1758 (1763)
_fpiomsg 1834 (1839)
_fpnofree 0930 (0932)
_fpomsg 1816 (1821)
_fptloop (1329) 1337

FPTrapPC 1756 1759 1776 1779 1794 1797 1814 1817 1832 1835
(2331)
_fpumsg 1778 (1783)
FreePage (0904) 1332
FreePg (0920) 1127
FreePRPT 0351 (1327)
FreeRM 0237 0239 0241 0243 (1125)
_fregtbl 1351 (1365)

- G -
G_CLK (0086) 0099 0112
GetByte 0379 0384 2048
GetUserB 2082 (2269)
GetUserH 1648 1904 (2286)
G_NAM (0087) 0100 0113
_gregtbl 1348 (1357)

- H -
Halt 0323 (1408) 1413
_haltmsg 0321 (0325)
HANG_SSW (0130)
_hexmsg 1489 1514 (1530)
_hextbl (1533)

- I -
I_Clock (2136)
_id0msg (1854)
_igname 0805 (0816)
_ignlen 0804 (0821)
_imtsmsg 2193 (2200)
InitGbl 0284 (0797)
INIT_U_FPH (0123)
INIT_U_FPL (0124)
INIT_U_PC (0120) 0437
INIT_U_PSW (0119) 0441
INIT_U_REG (0122)
InitUReg 0338 (1347)
INIT_U_SP (0121)
INIT_U_SSW (0118) 0439
I_Nmi (2122)
IntGrps 0278 (0782)
INTRAP_PC (0133)
INTRAP_PSW (0135)
INTRAP_SSW (0134)
INT_SSW (0129)
InvMTS (2192)
_iomsg 1908 (1917)
IO_PAGES (0024) 0171 0185 1088 2313 2315
IOUsage 0172 1086 (2312)
IOUseCnt 0186 1087 (2314)

_ipfmsg 1863 (1868)
I_Unexp (2104)
_IUnMsg 2108 (2114)

- L -
LoadPage 1191 1214 (1277)
LoadUData 1142 (1212)
LoadUGbl 1139 (1158)
LoadUPgm 0336 (1137)
LoadUProt 1138 (1261)
LoadUStk 1144 (1237)
LoadUText 1140 (1189)
_lpdone 1279 (1289)
_luddone 1215 (1225)
_lugfault 1162 (1171)
_lugmsg 1172 (1176)
_lupdone 1141 1143 (1145)
_lusdone 1240 (1250)
_lutdone 1192 (1199)

- M -
_mcflush (0383) 0386
_mcfound 0394 (0398)
_mcimsg 0402 (0406)
_mcinv 0392 (0401)
_mcloop (0389) 0396
_mcmdtbl (0412)
MonCmd 0304 (0374) 0381 0404
MonHalt (0319)
Monitor (0163)
MonLoop (0303) 0308
_monptbl (0312)
_mprompt 0375 (0409)

- N -
NBR_PTE (0019) 0836 1328
NewLine 1443 1460 1465 1470
NL (0143) 0348 0380 0385

- O -
OP_NOOP (0148)
OP_PUSHW (0149) 1650
OP_TRAPFN (0150) 1906

- P -
PAGE_LEN (0018) 0019 0798
PARM_BYTE (0139) 1438 1551 1973 1991 2111 2197
PARM_LONG (0140) 0346 0562 0846 0890 0949 1004 1580 1601 1636
1710 1735 1760 1780 1798 1818 1836 1865 1882 2126

```
PARM_NULL     (0138) 1173   1434   1684   1851   1910   1931
_pfmsg        2124   (2129)
PgTblInfo     (0569)
PGU_BS        (0046) 1065
PGU_DATA      (0050) 1213   2011
PGU_INIT      (0056)
PGU_IO        (0045) 1063
PGU_NONE      (0057) 0166   0166   0166   0166   0598   0867   0924   0982
PGU_PROT      (0055)
PGU_PTBL      (0048)
PGU_PTBLD     (0047)
PGU_RESV      (0054) 1111
PGU_SGBL      (0052)
PGU_STK       (0051) 1238   1656
PGU_TEXT      (0049) 1190
PGU_UGBL      (0053)
PGU_VECT      (0042)
_pimsg        1930   (1934)
_pmdone       1435   (1444)
_pmparm       1439   (1441)
PROT_NA       (0032) 0073
PROT_R        (0029) 0071
PROT_RE       (0031) 0063   0067
PROT_RW       (0028) 0065   0066   0068   0069   0070   0074
PROT_RWE      (0030) 0060   0061   0062   0064
PrtMsgV       0347   1382   1403   (1428) 1581   1992
PSW_C         0872   1051
PSW_MTS       2163   2195
PTBL_BS       (0064)
PTBL_DATA     (0068) 1220   2023
PTBL_INIT     (0074) 1281
PTBL_IO       (0063)
PTBL_NONE     (0075)
PTBL_PROT     (0073) 0912   1264
PTBL_PTBL     (0066)
PTBL_PTBLD    (0065)
PTBL_RESV     (0072)
PTBL_SGBL     (0070)
PTBL_STK      (0069) 1245   1668
PTBL_TEXT     (0067) 1194
PTBL_UGBL     (0071) 1164
PTBL_VECT     (0060)
PTDE_F        1033
PTDE_PTO      1035
PTE_F         0072   0075   0907   1040   1301
PTE_RA        0914   1044
PTE_ST        0916   1043
PTE_ST_CPRG   0074
PTE_ST_NCBS   0064   1081
PTE_ST_NCIO   0063   1079
```

PTE_ST_NCMM 0062 0065 0066 0070 0071 0073
PTE_ST_PVCB 0061 0068 0069
PTE_ST_SHWT 0060 0067
PutByte 0349 0378 1523 1527 2065 2085
PutHex32 (1493)
PutHex8 (1518)
PutHexB 1437 (1512)
PutHexW 1440 1459 1464 1469 (1487)
PutStr 0293 0322 0359 0376 0403 1384 1405 1431 1457 1462
1467 1490 1515 1946
_px32loop (1497)

- R -
_rdprmsg 1708 (1719)
ReadPage 1284
ResvRM 0194 0196 0198 0200 (1110)
RMAD_IVEC (0079) 0280
RMAD_SCVEC (0078) 0274
RMAD_SPT (0081)
RMAD_SPTD (0080) 0214
RMAD_TVEC (0077) 0268
RMAD_UPT (0083)
RMAD_UPTD (0082) 0231
RM_PAGES (0023) 0167 0181 1100 2305 2307
RMUsage 0168 1098 (2304)
RMUseCnt 0182 1099 (2306)

- S -
SC_ExDS (2008)
SC_Exit (1988)
SC_GetB (2045)
SCGrps 0272 (0762)
_scpsdone 2084 (2090)
_scpsloop (2080) 2088
SC_PutB (2061)
SC_PutS (2076)
SC_SSW (0128)
_SCUnMsg 1970 (1976)
SC_Unsupp (1966)
SetGPtde 0203 0220 (0505) 0509
SetGPte 0209 0226 (0543)
SetGrps 0205 0211 0222 0228 0269 0275 0281 (0477)
SetGVec 0267 0273 0279 (0673) 0679
setmap (0250)
SetPte (0834) 1166 1196 1222 1247 1266 1316 1670 2025
SetUBS 1066 1082 (1091)
SetUIO 1064 1080 (1085)
SetURM 1067 1083 (1097) 1126
SetUse 0862 (1062)
SetUseST 0917 0977 (1078)

SetWindow 1282 1302 (1313)
_sggunus 0599 (0612)
_sgloop (0480) 0492
_sgpfail 0550 (0559)
_sgploop (0548) 0556
_sgpmsg 0560 (0565)
SgptGet 0549 (0597)
_sgtest 0478 (0489)
SharePage (0974) 1161
SHF_PSW_MTS 2165 2196
SHF_PTE_RA 0551 0835 0915 0978
SHF_VA_PTDI 1030
SHF_VA_PTI 0101 0111 0114 0115 0116 1037 1641 2027
SP (0145) 0377
_spbadp (0991)
_spbaduc (0995)
_spbpmsg (1007)
_spbucmsg (1011)
_spdie (1001)
_spdone (0989)
_spfmsg (1015)
_spfree (0999)
SptdGrps 0202 (0516)
SptGrps 0208 (0620)
_spvpage (0844)
_spvpmsg (0849)
SSW_ECM 0118
SSW_EI 0118 0129 0288
SSW_IL 0118 0288 1550 1972 2110
SSW_IN 1550 1972 2110
SSW_M 0118 0127 0128 0129 0130 0252 1410
SSW_P 1630 1712 1737 1898 2217
SSW_TP 1949
SSW_U 0118
SSW_UU 1633 1715 1740 2271 2274 2288 2291
SVAD_GCLK (0099) 0812 2125 2138 2140
SVAD_GDAT (0098) 0099 0100 0799 1159
SVAD_GNAM (0100) 0292 0806
SVAD_MERR (0093) 1579 1600
SVAD_SPT (0095) 0210 1315
SVAD_SPTD (0094) 0204 1160
SVAD_STK (0102) 0262
SVAD_UPT (0097) 0227 0350 1165 1195 1221 1246 1265 1669 2024
SVAD_UPTD (0096) 0221
SVAD_WNDO (0101) 1283
SVPN_WNDO (0090) 0101 1314
SYS_EXDS (0036)
SYS_EXIT (0035)
SYS_GETB (0037)

SYS_PUTB (0038)
SYS_PUTS (0039)

- T -
T_DCMemErr (1562)
T_DPageFlt (1612)
T_DRdProt (1702)
T_DUMemErr (1593)
T_DWrtProt (1727)
_tfnmsg 1914 (1920)
T_FPDiv0 (1790)
T_FPInex (1752)
T_FPInvOp (1828)
T_FPOflow (1810)
T_FPUflow (1772)
T_ICMemErr (1563)
T_IDiv0 (1848)
T_IExProt (1878)
T_IllOp (1893)
_tiodone (1909) 1915
_tiotest 1901 (1905)
_tiotfn 1907 (1913)
_tiouser 1899 (1903)
T_IPageFlt (1861)
T_IUMemErr (1594)
T_PrivIns (1927)
Transfer (0450)
TransVA 0975 (1027)
TrapGrps 0266 (0686)
TRAP_SSW (0127)
_ttmsg 1945 (1954)
T_Trace (1942)
T_Unimpl (1545)
_tunmsg 1548 (1554)
_tvdone (1046) 1052
_tvfail 1034 1041 (1050)

- U -
U_BStkPg 1244 1640 1659 1664 2016 (2347)
U_DataLen 1217 2020 2022 (2341)
UMemErr (1598)
_umemsg 1599 (1604)
UptdGrps 0219 (0528)
UptGrps 0225 (0658)
U_RunTime 0340 0345 2141 2143 (2339)
UserR0 (1358)
U_StkLen 1242 1665 1667 (2343)
U_TDataPg 1219 1661 2014 2019 2026 (2345)
UVAD_DATA (0115)

```
UVAD_DATA       (0115)
UVAD_GCLK       (0112)
UVAD_GDAT       (0111) 0112   0113
UVAD_GNAM       (0113)
UVAD_STK        (0116) 0121
UVAD_TEXT       (0114) 0120
UVPN_DATA       (0107) 0115   1218
UVPN_GDAT       (0105) 0111   1163
UVPN_STK        (0108) 0116   1243
UVPN_TEXT       (0106) 0114   1193

- V -
VA_PTDI         1029
VA_PTI          1036
```

9.6.2 User Program Cross Reference

```
- B -
Buffer          0225    (0263)
BUFLEN          (0036) 0225    0264

- C -
CalcFibI        0058    (0120)
CalcFibR        0068    (0145)
_cdaloop        (0229) 0237
_cfiloop        (0127) 0134
_cfrloop        (0152) 0159
ClockStart      0057    0067    0205    (0252)
ClockStop       0060    0070    0204    (0256)
ConvertDA       0102    0207    (0224)

- D -
DPOne           0123    (0271)
DPZero          0121    (0268)

- E -
_etmsg          0202    (0213)
_exit           (0074) 0080

- F -
_fhigh          0174    (0178)
_fibimsg        0053    (0086)
FibR            0154    (0172) 0183    0187
_fibrmsg        0063    (0090)
FNumber         0048    0055    0065    (0248)

- G -
G_CLK           (0023) 0028
G_NAM           (0024) 0029

- M -
MAXFIB          (0035) 0101    0126    0158

- N -
NL              (0037) 0104    0209
_nopage         0047    (0076)
_nopmsg         0077    (0082)

- P -
PrintET         0061    0071    (0201)
PSW_FPE         0050
```

- S -

SignOn 0043 (0098)
_somsg 0099 (0108)
SP (0038)
StackAl 0150 0160 (0260)
Start (0042)
SYS_EXDS (0017) 0045
SYS_EXIT (0016) 0075
SYS_GETB (0018)
SYS_PUTB (0019) 0105 0210
SYS_PUTS (0020) 0054 0064 0078 0100 0103 0203 0208

- U -

UVAD_DATA (0031)
UVAD_GCLK (0028) 0056 0059 0066 0069
UVAD_GDAT (0027) 0028 0029
UVAD_GNAM (0029)
UVAD_STK (0032)

Chapter 10
Instruction Set

10.1 INTRODUCTION

The CLIPPER instruction set contains features derived from both the Reduced Instruction Set Computer (RISC) philosophy and from the Complex Instruction Set Computer (CISC) philosophy. The principal RISC features of the CLIPPER are its load/store architecture and hardwired instructions. A load/store machine is one in which arithmetic and logic operations are performed only on operands in registers; only the load and store operations (and branches) access memory directly. A load/store architecture permits the frequently used arithmetic and logical instructions to be executed in one clock cycle. All CLIPPER basic instructions are hardwired, that is, they are executed directly by the hardware and not by microcode; thus their execution speed is maximized. The main CISC feature is the Macro Instruction Unit, which contains higher level macro instructions, implemented as sequences of hardwired instructions, that perform complex operations.

The CLIPPER instruction set contains 101 hardwired instructions and 67 macro instructions. This chapter contains detailed descriptions of all CLIPPER instructions. The instructions are listed in alphabetical order with at least one page per instruction. Table 10.1 contains a summary of all the instructions.

Table 10.1 Alphabetical Listing of the Instructions

Mnemonic	Description	Page	Mnemonic	Description	Page
addd	Add Double Floating	10-7	movwp	Move Word to Processor Register	10-64
addi	Add Immediate	10-8	movws	Move Word to Single Floating	10-65
addq	Add Quick	10-9	muld	Multiply Double Floating	10-66
adds	Add Single Floating	10-10	muls	Multiply Single Floating	10-67
addw	Add Word	10-11	mulw	Multiply Word	10-68
addwc	Add Word with Carry	10-12	mulwu	Multiply Word Unsigned	10-69
andi	And Immediate	10-13	mulwux	Multiply Word Unsigned Extended	10-70
andw	And Word	10-14	mulwx	Multiply Word Extended	10-71
b*	Branch on Condition	10-15	negd	Negate Double Floating	10-72
bf*	Branch on Floating Exception	10-17	negs	Negate Single Floating	10-73
call	Call Subroutine	10-18	negw	Negate Word	10-74
calls	Call Supervisor	10-19	noop	No Operation	10-75
cmpc	Compare Characters	10-20	notq	Not Quick	10-76
cmpd	Compare Double Floating	10-21	notw	Not Word	10-77
cmpi	Compare Immediate	10-22	ori	Or Immediate	10-78
cmpq	Compare Quick	10-23	orw	Or Word	10-79
cmps	Compare Single Floating	10-24	popw	Pop Word	10-80
cmpw	Compare Word	10-25	pushw	Push Word	10-81
cnvds	Convert Double to Single Floating	10-26	restd*n*	Restore Registers f*n*: $0 \leq n \leq 7$	10-82
cnvdw	Convert Double to Word	10-27	restur	Restore User Registers	10-83
cnvrdw	Convert Rounding Double to Word	10-28	restw*n*	Restore Registers r*n*: $0 \leq n \leq 12$	10-84
cnvrsw	Convert Rounding Single to Word	10-29	ret	Return From Subroutine	10-85
cnvsd	Convert Single to Double Floating	10-30	reti	Return From Interrupt	10-86
cnvsw	Convert Single Floating to Word	10-31	roti	Rotate Immediate	10-87
cnvtdw	Convert Truncating Double to Word	10-32	rotl	Rotate Longword	10-88
cnvtsw	Convert Truncating Single to Word	10-33	rotli	Rotate Long Immediate	10-89
cnvwd	Convert Word to Double Floating	10-34	rotw	Rotate Word	10-90
cnvws	Convert Word to Single Floating	10-35	saved*n*	Save Registers f*n*: $0 \leq n \leq 7$	10-91
divd	Divide Double Floating	10-36	saveur	Save User Registers	10-92
divs	Divide Single Floating	10-37	savew*n*	Save Registers r*n*: $0 \leq n \leq 12$	10-93
divw	Divide Word	10-38	scalbd	Scale by Double Floating	10-94
divwu	Divide Word Unsigned	10-39	scalbs	Scale by Single Floating	10-95
initc	Initialize Characters	10-40	shai	Shift Arithmetic Immediate	10-96
loada	Load Address	10-41	shal	Shift Arithmetic Longword	10-97
loadb	Load Byte	10-42	shali	Shift Arithmetic Longword immediate	10-98
loadbu	Load Byte Unsigned	10-43	shaw	Shift Arithmetic Word	10-99
loadd	Load Double Floating	10-44	shli	Shift Logical Immediate	10-100
loadfs	Load Floating Status	10-45	shll	Shift Logical Longword	10-101
loadh	Load Halfword	10-46	shlli	Shift Logical Longword Immediate	10-102
loadhu	Load Halfword Unsigned	10-47	shlw	Shift Logical Word	10-103
loadi	Load Immediate	10-48	storb	Store Byte	10-104
loadq	Load Quick	10-49	stord	Store Double Floating	10-105
loads	Load Single Floating	10-50	storh	Store Halfword	10-106
loadw	Load Word	10-51	stors	Store Single Floating	10-107
modw	Modulus Word	10-52	storw	Store Word	10-108
modwu	Modulus Word Unsigned	10-53	subd	Subtract Double Floating	10-109
movc	Move Characters	10-54	subi	Subtract Immediate	10-110
movd	Move Double Floating	10-55	subq	Subtract Quick	10-111
movdl	Move Double Floating to Longword	10-56	subs	Subtract Single Floating	10-112
movld	Move Longword to Double	10-57	subw	Subtract Word	10-113
movpw	Move Processor Register to Word	10-58	subwc	Subtract Word with Carry	10-114
movs	Move Single Floating	10-59	trapfn	Trap On Floating Unordered	10-115
movsu	Move Supervisor to User	10-60	tsts	Test and Set	10-116
movsw	Move Single to Word	10-61	wait	Wait for Interrupt	10-117
movus	Move User to Supervisor	10-62	xori	Exclusive-OR Immediate	10-118
movw	Move Word	10-63	xorw	Exclusive-OR Word	10-119

The format of each instruction is described in detail in the following pages. Figure 10.1 below illustrates the information presented.

Instruction Name — mnemonic and descriptive name of the instruction.

Assembler Syntax — Bold means enter item exactly as shown. *Italics* means substitute correct value. Punctuation must be included as shown. See Table 10.2 for operands.

Description — the text that describes the operation of the instruction, functionally equivalent to Operation.

Operation — this is an equation describing the operation of the instruction. Tables 10.3 and 10.4 list operands used. When setting PSW flags, the value of the expression to the right of the "←" replaces the flag.

Exceptions — conditions under which an exception may occur. Many traps have trap enables in the PSW or SSW. See Chapter 2 for details.

Instruction Format — the class, bit patterns, and fields of the instruction. See Table 10.2 for operand and operator descriptions.

Instruction Example — an example of the source statement entered. See Table 10.6 for a description of the terms used.

ADDD **Add Double Floating** **ADDD**

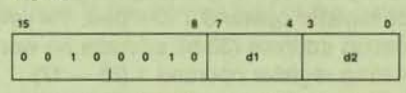
Syntax: **addd** *d1,d2*

Description: Add the double-precision contents of floating register d1 to the double-precision contents of floating register d2 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
d2 ← (d2) + (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register



Example:

```
loadd   dvalue,f0    # Load double floating value at dvalue into f0
loadd   (r2),f1      # Load double floating value at (r2) into f1
addd    f0,f1        # Add floating regs f0 and f1 put the result in f1
```

Figure 10.1 Instruction Description Example

Table 10.2 lists meanings of the operands used in the Syntax field of the instruction descriptions. The first character of the notation column specifies the operand's type and size:

b = byte	w = word	s = single-precision floating-point
h = halfword	l = longword	d = double-precision floating-point
		p = processor register

The second character of the notation column specifies the operand's field within the instruction and its location in the machine (immediate value, register, memory, etc.):

1 = R1	q = quick	a = address
2 = R2	i = immediate	b = byte

Table 10.2 Syntax Field Operands

Notation	Meaning
ba	Byte address (32-bit address on byte boundary)
bb	Unsigned byte data (0 — 256 decimal)
da	Double floating address (32-bit address on doubleword boundary)
d1	Double floating register operand 1 (f0 — f7)
d2	Double floating register operand 2 (f0 — f7)
ha	Halfword address (32 bit address on halfword boundary)
l1	Longword register operand 1 (even register pair r0,r1 — r14,r15)
l2	Longword register operand 2 (even register pair r0,r1 — r14,r15)
p1	Processor register operand 1 (0 = psw, 1 = ssw)
sa	Single floating address (32-bit address on word boundary)
s1	Single floating register operand 1 (f0 — f7)
s2	Single floating register operand 2 (f0 — f7)
wa	Word address (32 bit address on word boundary)
wi	Signed 16-bit immediate data (sign extended to 32-bits) or signed 32-bit immediate data
wq	Unsigned 4-bit quick data (zero extended to 32-bits)
w1	General register operand 1 (r0 — r15)
w2	General register operand 2 (r0 — r15)

Tables 10.2 and 10.3 list the meanings of the operands used in the Operation field of each description. Table 10.4 lists the meanings of the operators used in the Operation field of each description.

Table 10.3 Operation Field Operands

Notation	Meaning
C	PSW carry out flag
FP dest	Original floating-point destination
FP PC	Floating-point program counter
N	PSW integer negative flag
n	Number (usually register number)
PC	Program counter
r0 — r15	General registers 0 through 15
V	PSW integer overflow flag
Z	PSW integer zero flag

Table 10.4 Operation Field Operators

Notation	Meaning
rot	Rotate operator
sha	Shift arithmetic operator
shl	Shift logical operator
+	Add operator
–	Subtract operator or negate unary operator
×	Multiply operator
÷	Divide operator
mod	Modulus operator
~	Complement operator
=	Equal operator
≠	Not equal operator
←	Assignment operator
&	AND logical operator
|	OR logical operator
⊕	Exclusive-OR logical operator
( )	Contents of operand within
[ ]	Separators used to indicate value inside in a unit
< >	Value inside symbols indicates bit(s) of a register
:	Indicates a range of values

Tables 10.2 and 10.5 list meanings of the operands used in the Format field of the instruction descriptions.

Table 10.5 Format Field Operands

Notation	Meaning
addr mode	A four bit code for the type of addressing used
cond	Condition code for branch conditions (see Tables 10.7 and 10.8)
high	Most-significant portion of an address or value
low	Least-significant portion of an address or value
R1	Register field 1 within the instruction format
R2	Register field 2 within the instruction format

Table 10.6 lists the assembler instruction operands used in the Example field of the instruction descriptions. Assembler instruction operands are generally given in source, destination order, independent of their positions in the machine representation.

Table 10.6 Example Field Operands

r0 : r15	General registers. Even general registers address longword operands. sp, fp, and ap are synonyms for r15, r14, and r13. Not to be confused with R1 or R2, which are register fields within an instruction.
f0 : f7	Floating registers. Each register may contain either a single or double floating value
psw, ssw, ssw	Processor registers 0, 1 and 3
$*n*	Quick, byte or immediate value (decimal)
$0x*n*	Quick, byte or immediate value (hexadecimal)
n	Absolute address
n(m)	Relative or relative with displacement address. *n* may be 0 or absent.
[r*x*](rn)	Relative indexed address
n(pc)	PC relative address
.±*n*	PC relative address
[r*x*](pc)	PC indexed address
#	Indicates a comment field

ADDD — Add Double Floating — ADDD

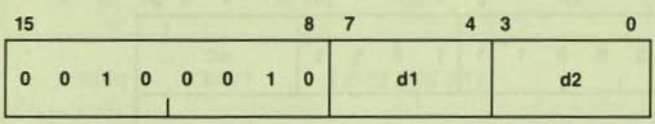
Syntax: **addd** *d1,d2*

Description: Add the double-precision contents of floating register d1 to the double-precision contents of floating register d2 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
d2 ← (d2) + (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register



Example:

```
loadd   dvalue,f0   # Load double floating value at dvalue into f0
loadd   (r2),f1     # Load double floating value at (r2) into f1
addd    f0,f1       # Add floating regs f0 and f1 and put the result in f1
```

ADDI — Add Immediate — ADDI

Syntax: **addi** *wi,w2*

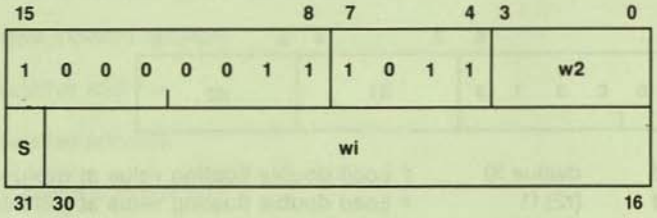
Description: Add the immediate value wi to the contents of general register w2 and put the result in w2. The operands may be signed or unsigned integers. Overflow is set if the operands (which are treated as signed integers) have the same sign and the result has the opposite sign.

Operation:
w2 ← (w2) + wi
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← integer carry out

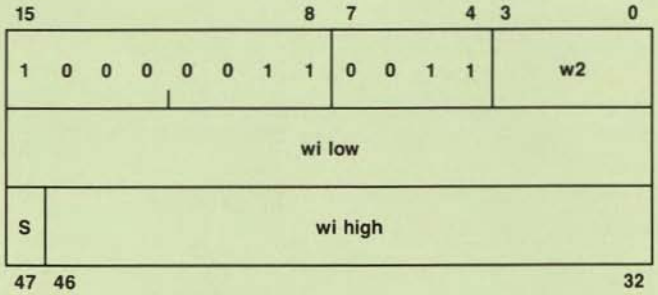
Traps: none

Format: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{15} - 1$, this format is used:



Example:

```
addi    $180,r1      # Add 16-bit value to r1
addi    $99999,r0    # Add 32-bit value to r0
addi    $-1,r2       # Add 0xffff to r2
```

ADDQ

Add Quick

ADDQ

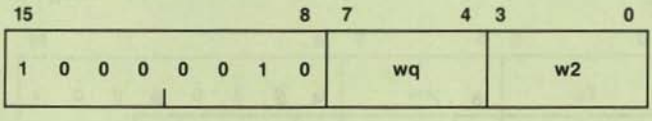
Syntax: **addq** *wq,w2*

Description: Add the unsigned quick value wq to the contents of general register w2 and put the result in w2. The contents of w2 may be a signed or unsigned integer. Overflow is set if the contents of w2 (which is treated as a signed integer) is positive and the result is negative.

Operation:
w2 ← (w2) + wq
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Quick



Example:
```
addq    $4,r1       # Add 4 to (r1)
```

10

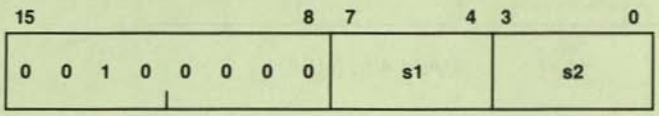
Syntax: **adds** *s1,s2*

Description: Add the single-precision contents of floating register s1 to the single-precision contents of floating register s2 and put the result in s2. On a trap, the PC and the original value in s2 can be obtained by using the **loadfs** instruction.

Operation:
s2 ← (s2) + (s1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register



Example:

```
loads   $value1,f0      # Load single floating value at $value1 into f0
loads   (r3),f2         # Load single floating value at (r3) into f2
adds    f2,f0           # Add floating regs f0 and f2 and put the result in f0
```

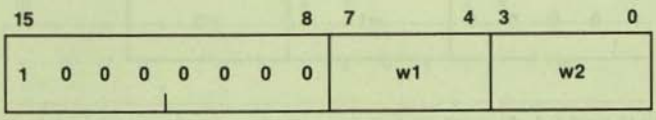
Syntax: **addw** *w1,w2*

Description: Add the contents of general register w1 to the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the operands (which are treated as signed integers) have the same sign and the result has the opposite sign.

Operation:
w2 ← (w2) + (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Register



Example:
```
addw    r12,r0      # Add contents of r12 and r0, put the result in r0
```

ADDWC — Add Word With Carry — ADDWC

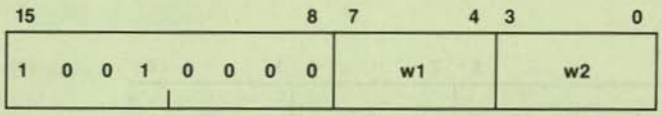
Syntax: **addwc** *w1,w2*

Description: Add the contents of general register w1 and the carry flag to the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers.

Operation:
w2 ← (w2) + (w1) + C
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Register



Example: Assume (r0,r1) and (r4,r5) contain two longwords and the operation to be performed is:

(r4,r5) ← (r4,r5) + (r0,r1).

```
addw    r0,r4    # Add low words, get carry
addwc   r1,r5    # Add high words using carry
```

Syntax: **andi** *wi,w2*

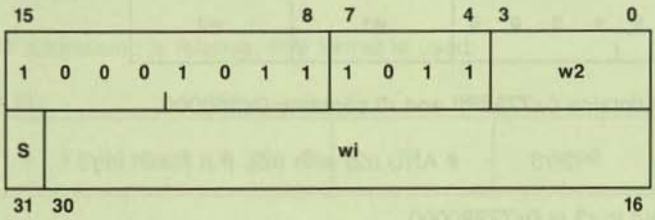
Description: Bitwise AND the immediate value wi with the contents of general register w2 and put the result in w2.

Operation:
w2 ← (w2) & wi
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

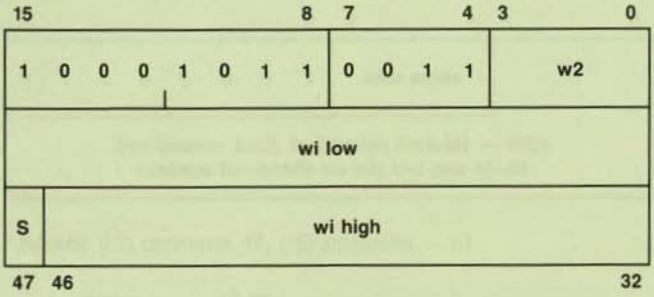
Traps: none

Format: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{31} - 1$, this format is used:



Examples: Assume r8 contains 0x001100ff.

```
andi    $0xffff,r8      # AND 32-bit value with r8
```

The result put in r8 is 0x000000ff.

Assume r2 contains 0xffff0000.

```
andi    $023,r2         # AND 16-bit value with r2
```

The result put in r2 is 0x00000000.

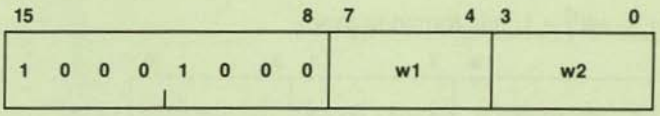
Syntax: **andw** *w1,w2*

Description: Bitwise AND the contents of general register w1 with the contents of general register w2 and put the result in w2.

Operation:
w2 ← (w2) & (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Example: Assume r2 contains 0x7788ffff and r3 contains 0xffff0000.

```
andw    r2,r3    # AND (r2) with (r3), put result in r3
```

The result put in r3 is 0x77880000.

Branch On Condition

B*

Syntax: **b*** *ha*

Description: If, for the selected condition cond, PSW flags C, V, Z, and, N match one of the patterns shown in Table 2-7, then the program branches to address ha; otherwise it does not branch. Cond is set by selecting one of the operands listed in Table 2-7.

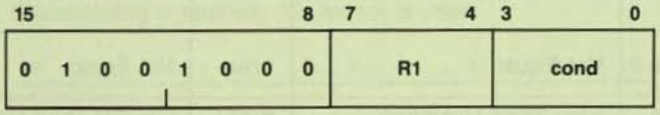
When a choice of mnemonics is shown, use the mnemonics beginning with bc if the conditions to be tested were set by a compare instruction. Use the mnemonics beginning with br if they were set by moves or logical instructions (those that set only N or Z).

Operation: if selected condition,
then PC ← ha

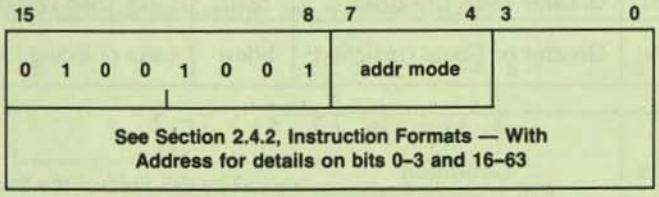
Traps: none

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Examples: Assume (r3) contains 17, (r6) contains – 19.

```
movw    r3,r4
brgt    label1      # Branches

cmpw    r3,r6
bcge    label2      # Branches

cmpw    r3,r6
bcgeu   label3      # Doesn't branch
```

Table 10.7 Integer Branch Conditions

cond	PSW Flags C V Z N	Name	Condition
0	X X X X	b	Branch always

cond	PSW Flags C V Z N	Name	Compare R1:R2	Name	Result R2:0
1	X 0 0 0 X 1 0 1	bclt	Less Than	brgt	Greater Than
2	X 0 X 0 X 1 0 1	bcle	Less or Equal	brge	Greater or Equal
3	X X 1 0	bceq	Equal	breq	Equal
4	X 0 0 1 X 1 X 0	bcgt	Greater Than	brlt	Less Than
5	X 1 X 0 X 0 0 1 X X 1 0	bcge	Greater or Equal	brle	Less or Equal
6	X X 0 X X X 1 1	bcne	Not Equal	brne	Not Equal
7	0 X 0 X	bcltu	Less Than Unsigned	brgtu	Greater Than Unsigned
8	0 X X X	bcleu	Less or Equal Unsigned	brgeu	Greater Than or Equal Unsigned
9	1 X X X	bcgtu	Greater Than Unsigned	brltu	Less Than Unsigned
A	1 X X X X X 1 X	bcgeu	Greater or Equal Unsigned	brleu	Less or Equal Unsigned

cond	PSW Flags C V Z N	Name	Condition
8	0 X X X	bnc	Not Carry
9	1 X X X	bc	Carry
B	X 1 X X	bv	oVerflow
C	X 0 X X	bnv	Not oVerflow
D	X X 0 1	bn	Negative
E	X X X 0	bnn	Not Negative
F	X X 1 1	bfn	Floating uNordered

Legend:
X = don't care

The R2 field of the **branch on condition** instruction selects the conditions on which to branch. When a choice of mnemonics is shown, use the ones beginning with **bc** if the condition codes to be tested were set by a compare instruction. Use the mnemonics beginning with **br** if they were set by move or logical instructions (those instructions that set only N or Z).

Syntax: **bf*** *ha*

Description: If the exceptions selected by the mnemonic are met, put ha in the PC. The cond field selects exceptions on which to branch as shown in Table 10.8.

Table 10.8 Floating Branch Conditions

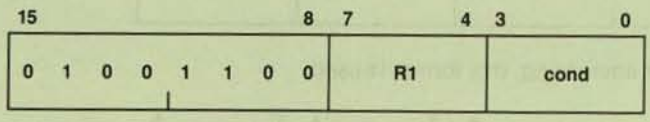
cond	Name	Exception
0	bfany	Floating ANY exception
1	bfbad	Floating BAD result
2–F		Reserved

Operation: if selection condition,
then PC ← ha

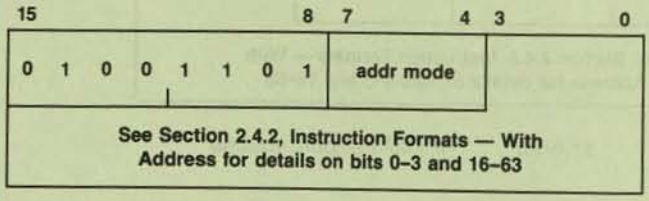
Traps: none

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example: Assume (f0) contains 1.0, (f1) contains 0.0, and there are no traps enabled.

```
divd    f0,f1     # Divide by zero
bfdz    label1    # branches
```

CALL — Call Subroutine — CALL

Syntax: **call** *w2,ha*

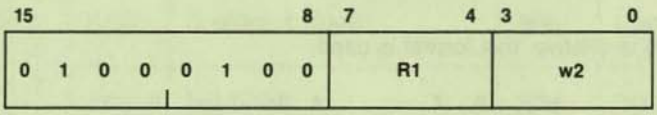
Description: The return address (the address of the instruction following the **call** instruction) is pushed onto the stack. The stack pointer is contained in w2. Control is then transferred to the specified address. On an exception, the stack pointer has not been modified.

Operation:
w2 ← (w2)−4
(w2) ← (PC)
PC ← ha

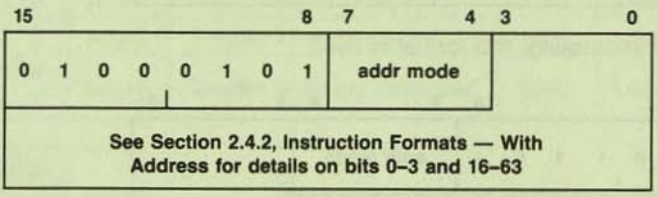
Traps:
Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:
```
call    sp,print    # Call the print routine
```

CALLS — Call Supervisor — CALLS

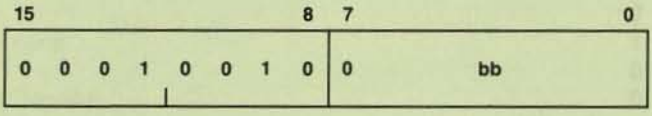
Syntax: **calls** *bb*

Description: Call the supervisor. Cause one of the 128 unique traps through the indicated supervisor trap vector bb. The current status registers (SSW, PSW and PC) are pushed on the supervisor's stack. The new PC and SSW are taken from the trap vector and the PSW is set to zero.

Operation: trap 400 + 8 × bb

Traps: none

Format: Control



Example:

```
calls    $6    # Invoke supervisor call # 6
```

Syntax: **cmpc**

Description: Compare a string of bytes. The string length is in r0, the address of the first string is in r1, and the address of the second string is in r2. On an exception, the registers are updated so that restarting the instruction compares the remaining portions of the strings. The bytes are signed extended to 32 bits, then compared as words.

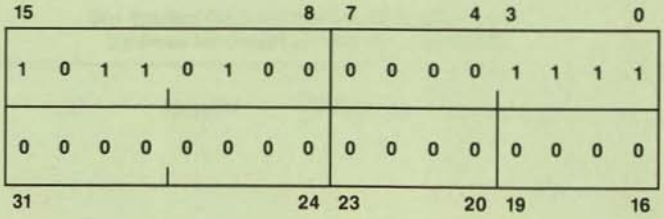
Operation:

```
while [(r0) ≠ 0] & [((r2)) = ((r1))],
        r0 ← (r0) – 1
        r1 ← (r1) + 1
        r2 ← (r2) + 1

if (r0) = 0,
  then N ← 0
       Z ← 1
       V ← 0
       C ← 0
  else temp ← (r2) – (r1)
       N ← (temp<31>)
       Z ← (temp) = 0
       V ← overflow
       C ← carry out
```

Traps:
Page fault
Read protect fault
Memory fault

Format: Macro



Example: Assume the string "ABCD" is at label **str1** and the string "ABXY" is at label **str2**.

```
loadq   $4,r0     # Load length into r0
loada   str1,r1   # Load addr of str1 into r1
loada   str2,r2   # Load addr of str2 into r2
cmpc              # Compare the strings
bcge    label1    # Does not branch, (r0) = 2,
                  # (r1) points to "C", str1 + 3
                  # (r2) points to "X", str2 + 3
```

Syntax: **cmpd** *d1,d2*

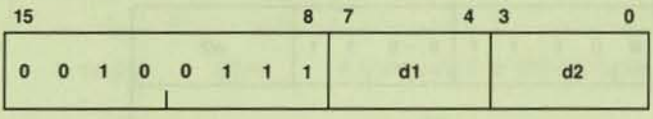
Description: Compare the double-precision contents of floating register d1 with the double-precision contents of floating register d2. Only the condition codes are affected.

NOTE
Although +0.0 and −0.0 are represented differently, they compare equal.

Operation:
(d2) − (d1)
N ← [[(d2) − (d1)] ≤ 0] OR [(d2),(d1) unordered]
Z ← [[(d2) − (d1)] = 0] OR [(d2),(d1) unordered]
V ← 0
C ← 0

Traps: none

Format: Register



Example:
```
cmpd    f1,f2    # Compare contents of floating regs 1 & 2
```

Syntax: **cmpi** *wi,w2*

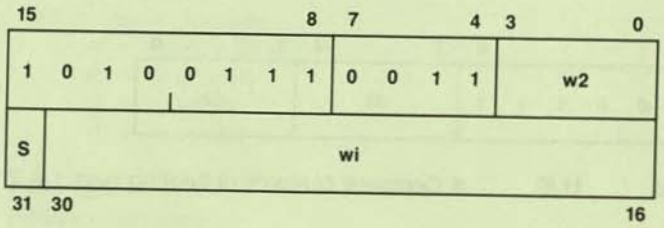
Description: Compare the immediate value wi with the contents of general register w2. Only the condition codes are affected.

Operation:
(w2) – wi
N ← [(w2) – wi] < 0
Z ← [(w2) – wi] = 0
V ← integer overflow
C ← integer carry out

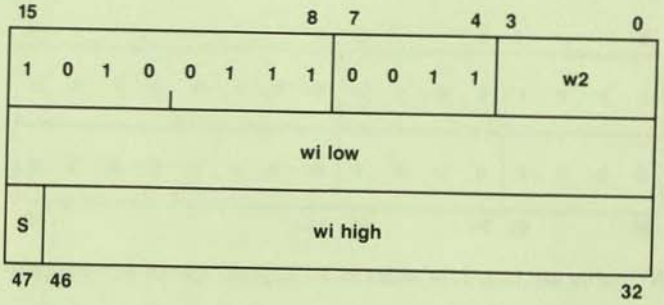
Traps: none

Format: Immediate

If $-2^{15} \leq$ wi $\leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq$ wi $\leq +2^{31} - 1$ or $-2^{15} \leq$ wi $\leq -2^{31} - 1$, this format is used:



Example:
```
cmpi    $0x1f,r2    # Compare 0x1f to the contents of r2
```

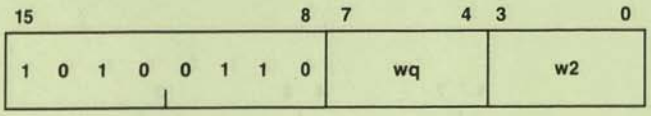
Syntax: **cmpq** *wq,w2*

Description: Compare the quick value wq with the contents of general register w2. Only the condition codes are affected.

Operation:
(w2) – wq
N ← [(w2) – wq] < 0
Z ← [(w2) – wq] = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Quick



Example:
```
cmpq    $0,r3    # Compare 0 to the contents of r3
```

Syntax: **cmps** *s1,s2*

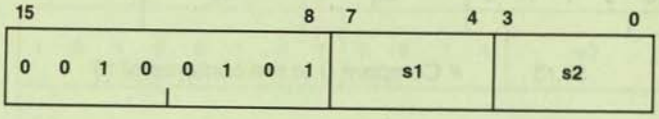
Description: Compare the single-precision contents of floating register s1 with the single-precision contents of floating register s2. Only the condition codes are affected.

NOTE
Although +0.0 and −0.0 are represented differently, they compare equal.

Operation:
(s2) − (s1)
N ← [[(d2) − (d1)] < 0] OR [(d2),(d1) unordered]
Z ← [[(d2) − (d1)] = 0] OR [(d2),(d1) unordered]
V ← 0
C ← 0

Traps: none

Format: Register



Example:
```
cmps    f0,f3    # Compare the contents of f0 to the contents of f3
```

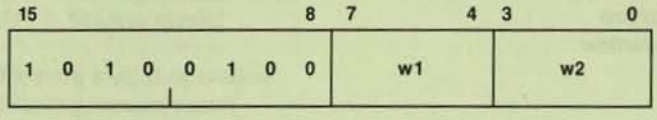
Syntax: **cmpw** *w1,w2*

Description: Compare the contents of general register w1 with the contents of general register w2. Only the condition codes are affected.

Operation:
(w2) – (w1)
N ← [(w2) – (w1)] < 0
Z ← [(w2) – (w1)] = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Register



Example:
```
cmpw    r0,r2    # Compare the contents of r0 to the contents of r2
```

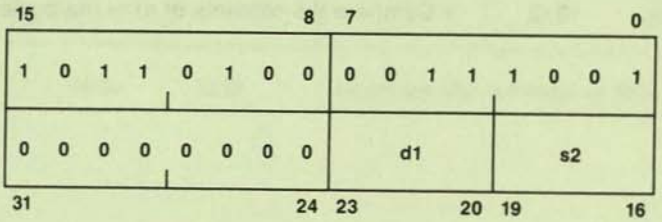
Syntax: **cnvds** *d1,s2*

Description: Convert the double-precision contents of floating register d1 to single-precision and put the result in floating register s2. On a trap, the PC and the original value in s2 can be obtained by using the **loadfs** instruction. The source and destination registers may be the same register.

Operation:
s2 ← (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Macro



Example:

```
cnvds   f0,f2   # Convert double in f0 to single, put in f2
cnvds   f1,f1   # Convert double in f1 to single
```

CNVDW — Convert Double Floating to Word — CNVDW

Syntax: **cnvdw** *d1,w2*

Description: Convert the double-precision contents of floating register d1 to a signed integer using the IEEE rounding mode given in the PSW and put the result in general register w2.

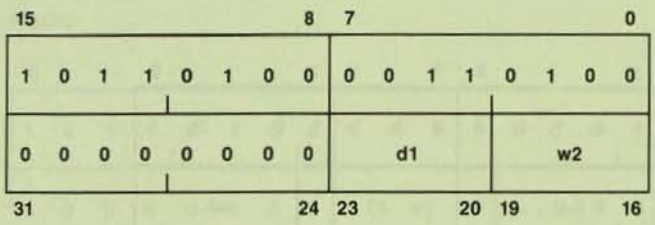
Attempting to convert $\pm\infty$, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (d1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro



Example:
```
cnvdw    f1,r2    # Convert double in f1 to integer, put in r2
```

CNVRDW Convert Rounding Double Floating to Word CNVRDW

Syntax: **cnvrdw** *d1,w2*

Description: Convert the double-precision contents of floating register d1 to the nearest signed integer by adding a properly signed 0.5, truncating toward 0.0, and putting the result in general register w2. This is the round operation in ISO Pascal, regardless of the IEEE rounding mode given in the PSW, and differs from the IEEE round-to-nearest mode.

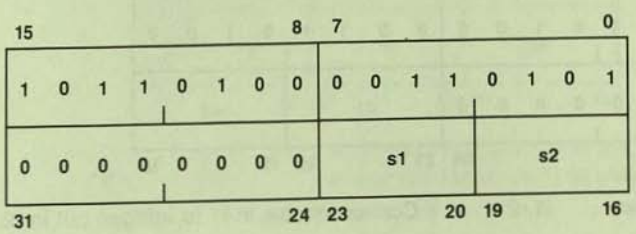
Attempting to convert $\pm\infty$, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number for which the truncation portion of the operation is inexact causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation: w2 ← (d1)
FX ← floating inexact result
FI ← floating invalid

Traps: Floating invalid operation
Floating inexact result

Format: Macro



Example:
```
cnvrdw    f0,r2    # Convert double in f0 to integer, put in r2
```

CNVRSW — Convert Rounding Single Floating to Word — CNVRSW

Syntax: **cnvrsw** *s1,w2*

Description: Convert the single-precision contents of floating register s1 to the nearest signed integer by adding a properly signed 0.5, truncating toward 0.0, and putting the result in general register w2. This is the round operation in ISO Pascal, independent of the IEEE rounding mode in the PSW, and differs from the IEEE round to nearest mode.

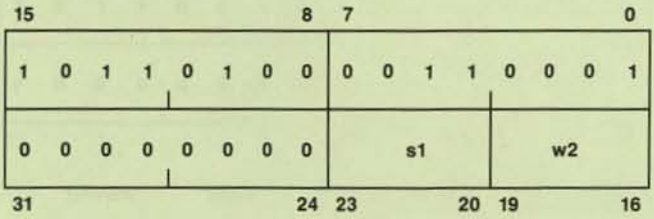
Attempting to convert $\pm\infty$, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number for which the truncation portion of the operation is inexact causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (s1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro



Example:
```
cnvrsw      f1,r0       # Convert single in f1 to integer, put in r0
```

CNVSD — Convert Single to Double Floating — CNVSD

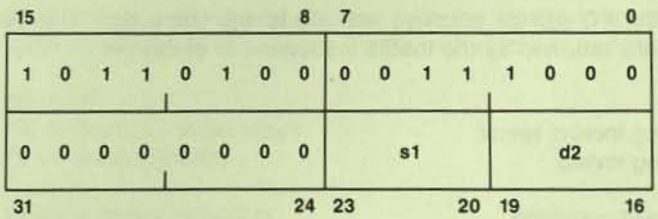
Syntax: **cnvsd** *s1,d2*

Description: Convert the single-precision contents of floating register s1 to double-precision and put the result in floating register d2. The source and destination registers may be the same register.

Operation: d2 ← (s1)
FI ← floating invalid

Traps: Floating invalid operation

Format: Macro



Example:

```
cnvsd    f0,f2    # Convert single in f2 to double, put in f2
```

Syntax: **cnvsw** *s1,w2*

Description: Convert the single-precision contents of floating register s2 to a signed integer using the IEEE rounding mode given in th PSW, and put the result in general register w2.

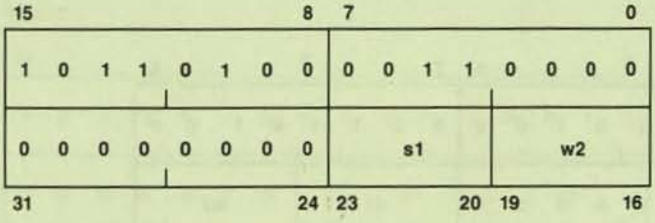
Attempting to convert $\pm\infty$, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation: w2 ← (s1)
FX ← floating inexact result
FI ← floating invalid

Traps: Floating invalid operation
Floating inexact result

Format: Macro



Example:
```
cnvsw    f0,r3    # Convert single in f0 to integer, result in w2
```

CNVTDW Convert Truncating Double Floating to Word CNVTDW

Syntax: **cnvtdw** *d1,w2*

Description: Convert the double-precision contents of floating register d1 to a signed integer by truncating toward 0.0 and putting the result in general register w2. This is the INT operation in ANSI FORTRAN 77 and the trunc operation in ISO Pascal, regardless of the setting of the IEEE rounding mode in the PSW.

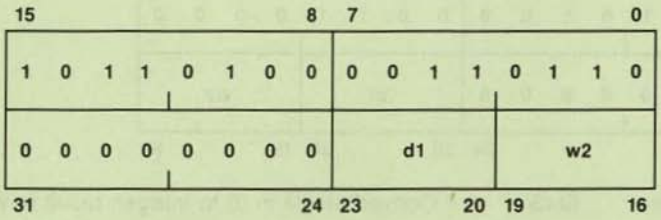
Attempting to convert $\pm\infty$, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation: w2 ← (d1)
FX ← floating inexact result
FI ← floating invalid

Traps: Floating invalid operation
Floating inexact result

Format: Macro



Example:
```
cnvtdw    f1,r1    # Truncate double in f1 to integer, put in r2
```

CNVTSW — Convert Truncating Single Floating to Word — CNVTSW

Syntax: **cnvtsw** *s1,w2*

Description: Convert the single-precision contents of floating register s1 to a signed integer by truncating toward 0.0 and putting the result in general register w2. This is the INT operation in ANSI FORTRAN 77 and the trunc operation in ISO Pascal, regardless of the setting of the IEEE rounding mode in the PSW.

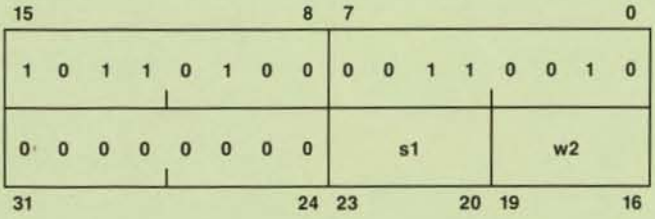
Attempting to convert $\pm\infty$, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (s1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro



Example:
```
cnvtsw    f0,r2    # Truncate f0 to integer, store in r2
```

CNVWD

Convert Word to Double Floating

CNVWD

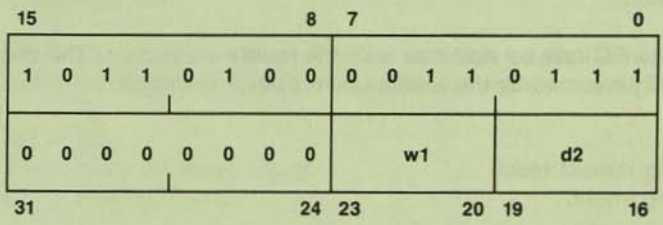
Syntax: **cnvwd** *w1,d2*

Description: Convert the contents of general register w1 to double-precision and put the result in floating register d2.

Operation: d2 ← (w1)

Traps: none

Format: Macro



Example:

```
cnvwd    r1,f0    # Convert integer in r1 to double, result in f0
```

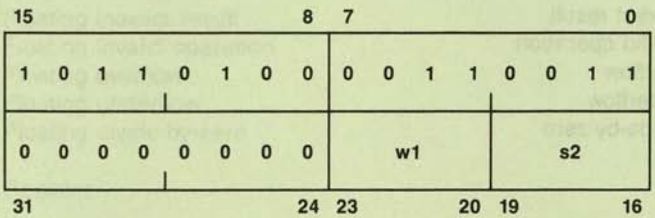
Syntax: **cnvws** *w1,s2*

Description: Convert the contents of general register w1 to single-precision floating-point and put the result in floating register s2. If the conversion is not exact, the result is rounded according to the IEEE rounding mode in the PSW, and floating inexact result is signalled.

Operation: s2 ← (w1)
FX ← floating inexact result

Traps: Floating inexact result

Format: Macro



Example:
```
cnvws   r3,f2   # Convert integer in r3 to single, result in f2
```

DIVD — Divide Double Floating — DIVD

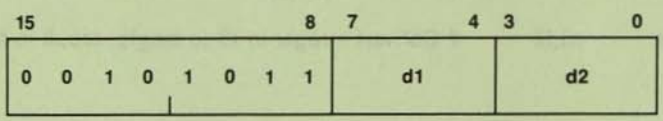
Syntax: **divd** *d1,d2*

Description: Divide the double-precision contents of floating register d2 by the double-precision contents of floating register d1 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
d2 ← (d2) ÷ (d1)
FX ← floating inexact result
FU ← floating underflow
FD ← floating divide-by-zero
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow
Floating divide-by-zero

Format: Register



Example:
```
divd    f2,f0    # Divide double in f0 by double in f2, put result in f0
```

Divide Single Floating

DIVS

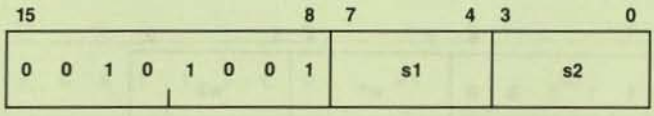
Syntax: **divs** *s1,s2*

Description: Divide the single-precision contents of floating register s2 by the single-precision contents of floating register s1 and put the result in s2. On a trap, the PC and the original value in s2 can be obtained by using the **loadfs** instruction.

Operation:
(s2) ← (s2) ÷ (s1)
FX ← floating inexact result
FU ← floating underflow
FD ← floating divide-by-zero
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow
Floating divide-by-zero

Format: Register



Example:
```
divs    f1,f2    # Divide single in f2 by single in f1, put result in f2
```

Syntax: **divw** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the *quotient* in w2. The operands are treated as signed integers. The quotient is positive if the signs of the divisor and dividend are the same, and negative if they are different. Overflow is set if the largest negative number (-2^{31}) is divided by -1.

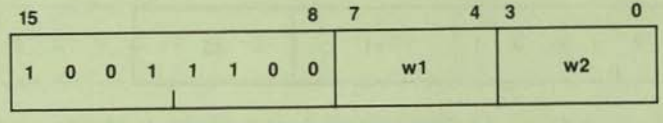
NOTE

For the same dividend and divisor, if the quotient returned by **divw** is multiplied by the divisor and then added to the remainder returned by **modw,** the result would be the original dividend.

Operation:
w2 ← (w2) ÷ (w1)
N ← 0
Z ← 0
V ← integer overflow
C ← 0

Traps: Divide-by-zero

Format: Register



Example:
```
divw    r10,r4    # Divide (r4) by (r10), put result in r4
```

DIVWU — Divide Word Unsigned — DIVWU

Syntax: **divwu** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the quotient in w2. The operands are treated as unsigned integers. Since the quotient is always positive, overflow cannot occur.

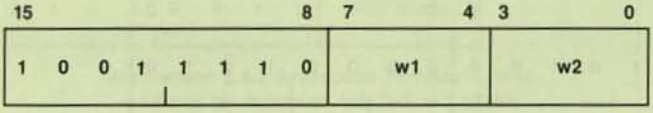
NOTE

For the same dividend and divisor, if the quotient returned by **divwu** and the remainder returned by **modwu** are multiplied, the result would be the original dividend.

Operation:
w2 ← (w2) ÷ (w1)
N ← 0
Z ← 0
V ← 0
C ← 0

Traps: Divide-by-zero

Format: Register



Example:
```
divwu    r0,r5    # Divide (r5) by (r0), put result in r5
```

10

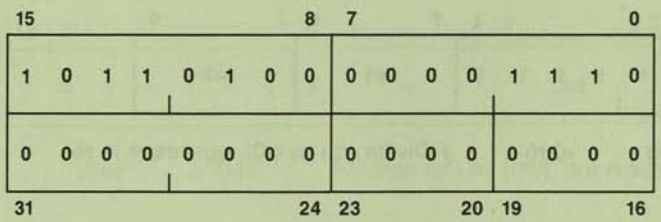
Syntax: **initc**

Description: Initialize a byte, halfword, word, or single-precision floating string with a constant value. The length of the string (in bytes) is in the r0 and the address is in r1. The pattern to be stored is in r2. For word and single-precision floating strings, r2 contains the initial value. For halfword strings, both halfwords of r2 must contain the initial value. For byte strings, all four bytes of r2 must contain the desired value. If a trap occurs, the registers are updated so that restarting the instruction initializes the remaining portion of the string.

Operation:
while (r0) ≠ 0,
(r1) ← (r2<7:0>)
r0 ← (r0) – 1
r1 ← (r1) + 1
r2 ← (r2) ROT – 8

Traps:
Page fault
Write protect fault
Memory fault

Format: Macro



Example: Set the 17 bytes beginning at label **str** to 19.

```
loadi   $17,r0          # String length
loada   str,r1          # String address
loadi   $0x13131313,r2  # 19 decimal = 13 hex
initc                   # Initialize characters
```

Syntax: **loada** *ba,w2*

Description: Load the memory address ba into general register w2.

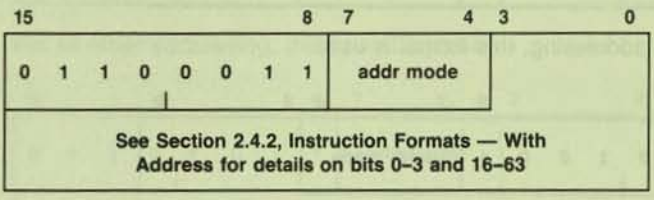
Operation: w2 ← ba

Traps: none

Formats: Address

If addressing is relative, this format is used:

For all other addressing, this format is used:



Examples:

```
loada   value,r0        # Address of value into r0
loada   8(sp),fp        # (sp) + 8 into fp
loada   addr(r4),r0     # (r4) + addr into r0
loada   [r1](r2),r3     # 3 operand addr that sets no flags
```

Syntax: **loadb** *ba,w2*

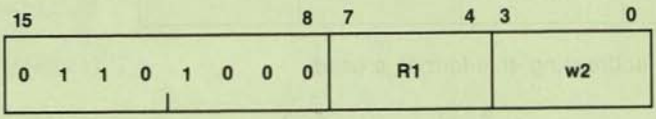
Description: Load the byte at memory address ba, sign-extended, into the least-significant byte of general register w2.

Operation: w2 ← (ba)

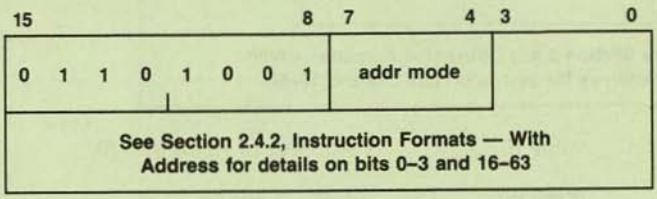
Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
loadb   3(r7),r2    # Load byte at 3(r7) into r2
```

Syntax: **loadbu** *ba,w2*

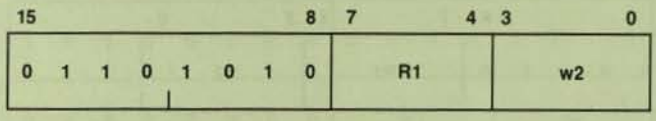
Description: Load the byte at memory address ba, zero-extended, into the least-significant byte of general register w2.

Operation: w2 ← (ba)

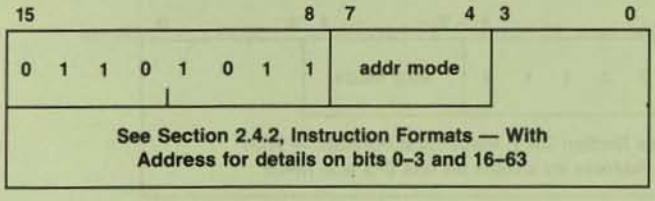
Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:
```
loadbu      4(r5),r2      # Load byte at 4(r5) into r2
```

LOADD — Load Double Floating — LOADD

Syntax: **loadd** *da,d2*

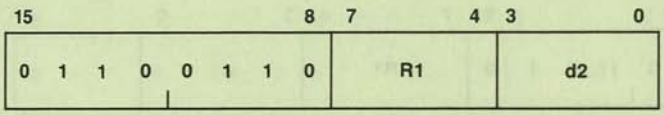
Description: Load the double-precision floating-point value at memory addresses da and da + 1 into floating register d2.

Operation: d2 ← (da)

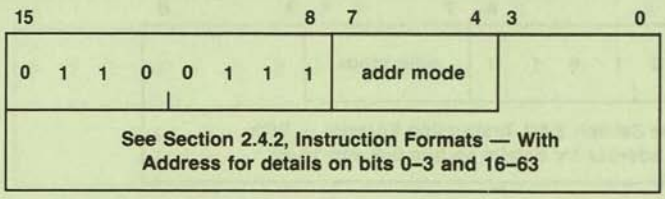
Traps:
Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Examples:

```
loadd   addr,f2      # Load double floating value at addr into f2
loadd   4(r2),f2     # Load double floating value at address specified by 4(r2)
                     # into f2
```

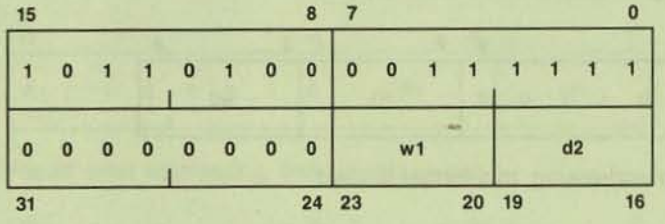
Syntax: **loadfs** *w1,d2*

Description: Load the floating status following a floating trap. The PC of the offending floating instruction is put in general register w1. The original value of the destination register is put in floating register d2. This information allows a trap handler to determine the original operation and its operands.

Operation: w1 ← (FP PC)
d2 ← (FP dest)

Traps: none

Format: Macro



Example:
```
loadfs    r1,f2    # Load FP PC into r1, and original destination into f2
```

Syntax: **loadh** *ha,w2*

Description: Load the halfword at memory address ha, sign-extended, into the least-significant halfword of general register w2.

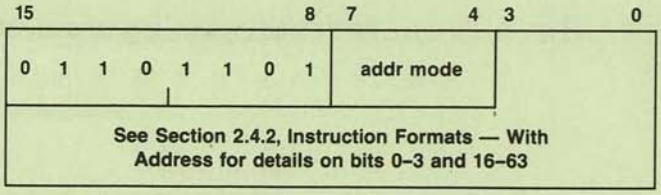
Operation: w2 ← (ha)

Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

For all other addressing, this format is used:



Example:

```
loadh    (r15),r12    # Load the halfword at (r15) into r12
```

LOADHU — Load Halfword Unsigned — LOADHU

Syntax: **loadhu** *ha,w2*

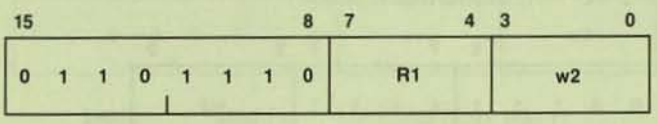
Description: Load the halfword at memory address ha, zero-extended, into the least-significant halfword of general register w2.

Operation: w2 ← (ha)

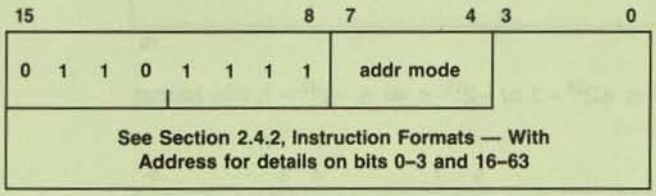
Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
loadhu      new,r11      # Load the halfword at new into r11
```

Syntax: **loadi** *wi,w2*

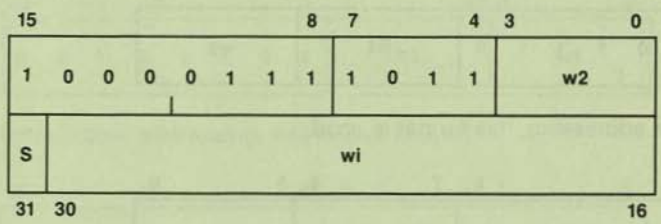
Description: Load the immediate value wi, sign-extended, into general register w2.

Operation:
w2 ← wi
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

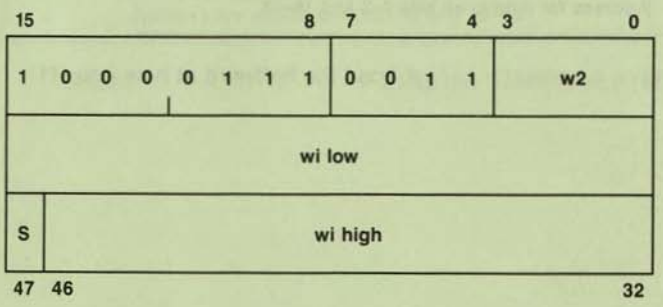
Traps: none

Format: Immediate

If $-2^{15} \leq$ wi $\leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq$ wi $\leq +2^{31} - 1$ or $-2^{15} \leq$ wi $\leq -2^{15} - 1$, this format is used:



Example:
```
loadi   $21,r2    # Load 21 into r2
```

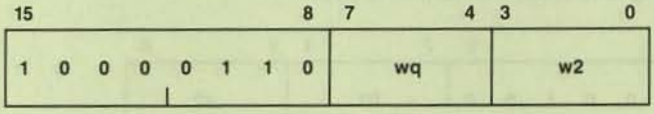
Syntax: **loadq** *wq,w2*

Description: Load the quick value wq, zero-extended, into general register w2.

Operation:
w2 ← wq
N ← 0
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Quick



Example:
```
loadq    $0xf,r1    # Load f hex into r1
```

Syntax: **loads** *sa,s2*

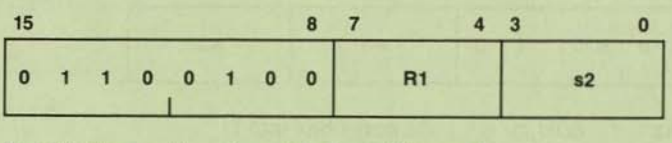
Description: Load the single-precision floating-point value at memory address sa into floating register s2.

Operation: s2 ← (sa)

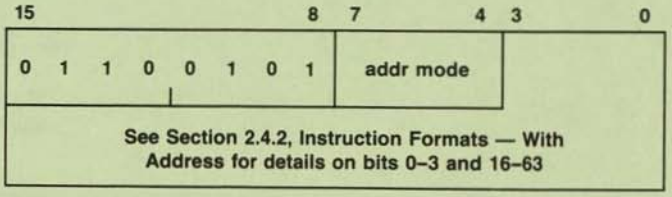
Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
loads    clock,f2    # Load single floating value at clock into f2
loads    (r3), f3    # Load single floating value at (r3) into f3
```

Syntax: **loadw** *wa,w2*

Description: Load the contents of memory address wa into general register w2.

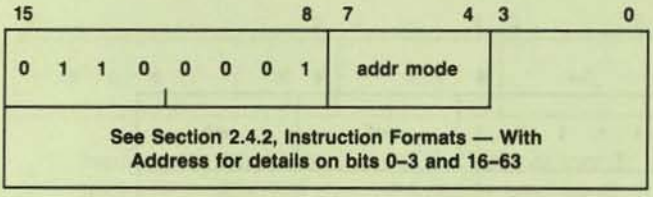
Operation: w2 ← (wa)

Traps: Page fault
Read protect fault
memory fault

Format: Address

If addressing is relative, this format is used:

For all other addressing, this format is used:



Example:

```
loadw    abc,r2        # Load word at abc into r2
loadw    $0xfca2,r1    # Load word at address fca2 hex into r1
```

Syntax: **modw** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the *remainder* in w2. The operands are treated as signed integers. The quotient is positive if the signs of the divisor and dividend are the same, and negative if they are different. If the remainder is non-zero, it has the same sign as the dividend. (Overflow is set if the largest negative number (-2^{31}) is divided by -1.)

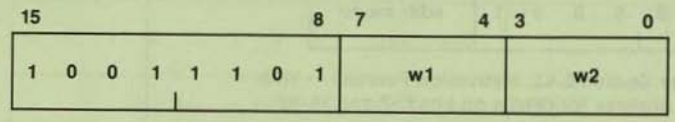
NOTE

For the same dividend and divisor, if the quotient returned by **divw** is multiplied by the divisor and then added to the remainder returned by **modw**, the result would be the original dividend.

Operation:
w2 ← (w2) MOD (w1)
N ← 0
Z ← 0
V ← integer overflow
C ← 0

Traps: Divide-by-zero

Format: Register



Example:

```
loadi   $1234,r2    # Load 1234 into r2
loadq   $11,r1      # Load 11 into r1
modw    r1,r2       # Divide r2 by r1, save remainder in r2
```

MODWU — Modulus Word Unsigned — MODWU

Syntax: **modwu** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the remainder in w2. The operands are treated as unsigned integers. If the remainder is non-zero, then it is positive.

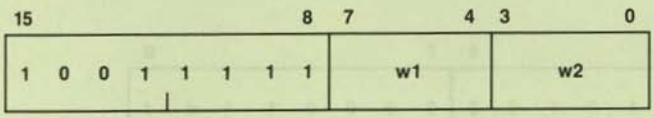
NOTE

For the same dividend and divisor, if the quotient returned by **divwu** and the remainder returned by **modwu** are multiplied, the result would be the original dividend.

Operation:
w2 ← (w2) MOD (w1)
N ← 0
Z ← 0
V ← 0
C ← 0

Traps: Divide-by-zero

Format: Register



Example:

```
loadi   $0xfffe,r5      # Load fffe hex into r5
loadq   $0x11,r0        # Load 11 hex into r0
modw    r0,r5           # Divide r5 by r0, save remainder in r5
```

Syntax: **movc**

Description: Move a string of bytes. The length of the strings is in r0, the address of the source string is in r1, and the address of the destination string is in r2. On an exception, the registers are updated so that restarting the instruction moves the remaining portion of the string.

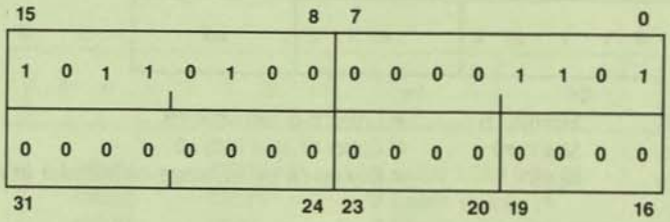
The source and destination strings may not overlap. The **initc** instruction should be used to initialize memory or the source string may be moved in blocks.

Operation:

```
while (r0) ≠ 0
      (r2)← ((r1))
       r0 ← (r0) – 1
       r1 ← (r1) + 1
       r2 ← (r2) + 1
```

Traps:
Page fault
Read protect fault
Write protect fault
Memory fault

Format: Macro



Example: Move a string of 124 characters at label **string1** to **string2.**

```
loadi   $124,r0      # Load string length into r0
loada   string1,r1   # Load address of source string into r1
loada   string2,r2   # Load address of destination string into r2
movc                 # Move the characters
```

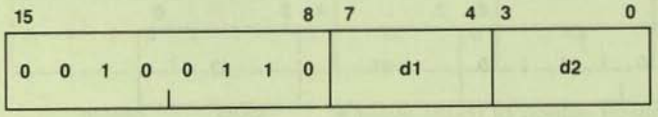
Syntax: **movd** *d1,d2*

Description: Move the double-precision contents of floating register d1 to floating register d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation: d2 ← (d1)

Traps: none

Format: Register



Example:

```
movd    f1,f2    # Move (f1) to f2
```

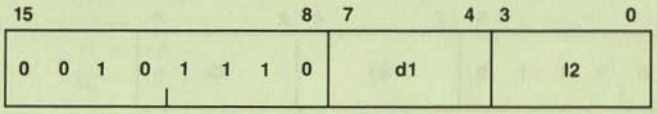
Syntax: **movdl** *d1,l2*

Description: Move the double-precision contents of floating register d1 to longword register pair l2 without conversion.

Operation: l2 ← (d1)

Traps: none

Format: Register



Example:
```
movdl    f1,r2    # Move (f1) to reg pair (r2,r3)
```

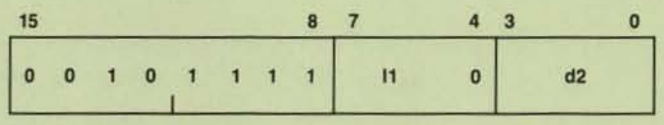
Syntax: **movld** *l1,d2*

Description: Move the contents of longword register pair l1 to floating register d2 without conversion.

Operation: (d2) ← (l1)

Traps: none

Format: Register



Example:

```
movld    r0,f2      # Move (r0,r1) to double floating reg f2
```

Syntax: **movpw** *p1,w2*

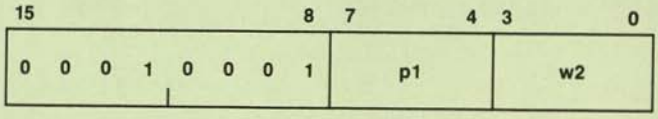
Description: Move the contents of processor register p1 to general register w2. The p1 value is interpreted as follows:

p1	Name	Meaning
0	PSW	Program status word
1	SSW	Supervisor status word
2-15	—	(Reserved)

Operation: w2 ← (p1)

Traps: none

Format: Register



Examples:

```
movpw   psw,r2   # Move (PSW) to r2
movpw   ssw,r2   # Move (SSW) to r2
```

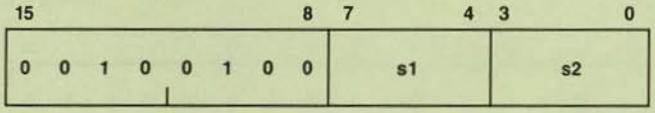
Syntax: **movs** *s1,s2*

Description: Move the single-precision contents of floating register s1 to floating register s2.

Operation: s2 ← (s1)

Traps: none

Format: Register



Example:

```
movs    f1,f2    # Move (f1) to f2
```

Syntax: **movsu** *w1,w2*

Description: Move the contents of supervisor general register w1 to user general register w2. A privileged instruction trap occurs if this instruction is executed in user mode.

Operation: $w2_{usr} \leftarrow (w1)_{sup}$

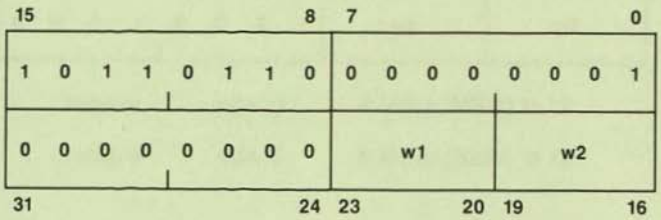
$N \leftarrow (w2<31>)_{usr}$

$Z \leftarrow (w2)_{usr} = 0$

$V \leftarrow 0$
$C \leftarrow 0$

Traps: Privileged instruction

Format: Macro



Example:

```
movsu     r0,r11     # Move (r0) to r11
```

MOVSW — Move Single Floating to Word — MOVSW

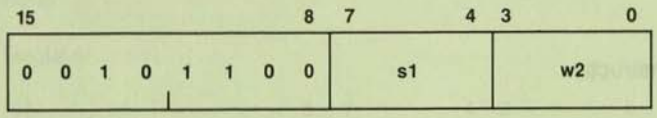
Syntax: **movsw** *s1,w2*

Description: Move the single-precision contents of floating register s1 to general register w2 without conversion.

Operation: w2 ← (s1)

Traps: none

Format: Register



Example:

```
movsw    f1,r2    # Move (f1) to r2
```

Syntax: **movus** *w1,w2*

Description: Move the contents of user general register w1 to supervisor general register w2.

Operation: $w2_{sup} \leftarrow (w1)_{usr}$

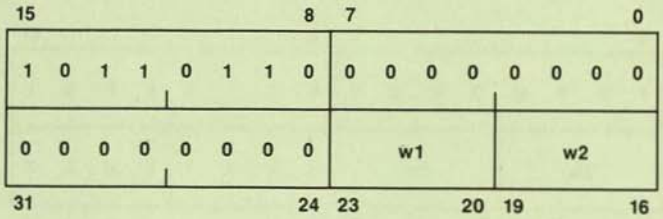
$N \leftarrow (w2<31>)_{sup}$

$Z \leftarrow (w2)_{sup} = 0$

$V \leftarrow 0$
$C \leftarrow 0$

Traps: Privileged instruction

Format: Macro



Example:
```
movus    r1,r2    # Move (r1) to r2
```

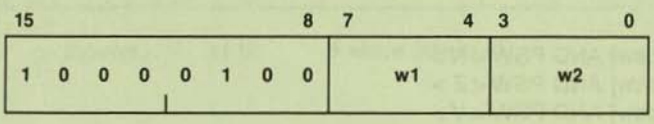
Syntax: **movw** *w1,w2*

Description: Move the contents of general register w1 to general register w2.

Operation:
w2 ← (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Example:
```
movw    r3,r0    # Move (r3) to r0
```

MOVWP — Move Word to Processor Register — MOVWP

Syntax: **movwp** *w2,p1*

Description: Move the contents of general register w2 to processor register p1. The p1 value is interpreted as follows:

p1	Name	Meaning
0	PSW	Program status word
1	SSW	Supervisor status word
3	SSWF	Supervisor status word fast
2,4–15	—	(Reserved)

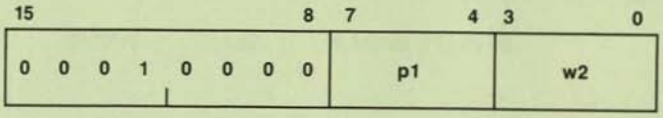
If p1 represents the PSW, then the condition codes are also set. Attempting to modify the SSW in user mode will cause a **noop.**

Operation:

p1 ← (w2)
N ← [p1 = psw] AND PSW<N>
Z ← [p1 = psw] AND PSW<Z>
V ← [p1 = psw] AND PSW<V>
C ← [p1 = psw] AND PSW<C>

Traps: none

Format: Register



Examples:

```
movwp    r2,psw    # Move (r2) to PSW
```

The following example only works in supervisor mode:

```
movwp    r2,ssw    # Move (r2) to SSW
```

MOVWS — Move Word to Single Floating — MOVWS

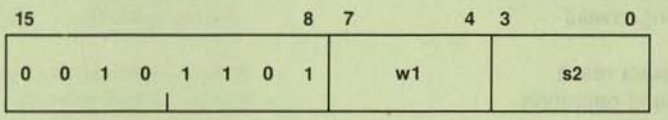
Syntax: **movws** *w1,s2*

Description: Move the contents of general register w1 to floating register s2 without conversion.

Operation: s2 ← (w1)

Traps: none

Format: Register



Example:

```
movws    r1,f2    # Move (r1) to f2
```

10

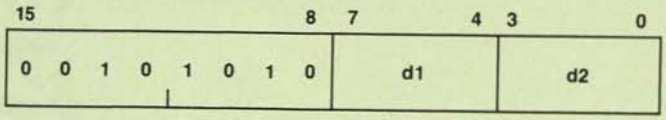
Syntax: **muld** *d1,d2*

Description: Multiply the double-precision contents of floating register d2 by the double-precision contents of floating register d1 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
d2 ← (d2) x (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register



Example: Assume fpval1 contains 1239237.1234 and fpval2 contains 8989.44334.

```
loadd    fpval1,f1    # Load double floating value
loadd    fpval2,f2    # Load double floating value
muld     f1,f2        # Multiply the values
```

The result is 11140051905.62889.

MULS — Multiply Single Floating — MULS

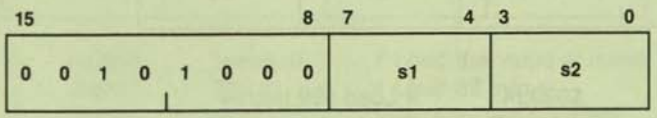
Syntax: **muls** *s1,s2*

Description: Multiply the single-precision contents of floating register s2 by the single-precision contents of floating register s1 and put the result in s2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
s2 ← (s2) × (s1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register



Example:

```
            loadi     $0xf8ccd,r0     # Load r0 with value
            movws     r0,f0           # Transfer to f0
            loadi     $0xc27dd,r0     # Load r0 with value
            movws     r0,f1           # Transfer to f1
            muls      f0,f1           # Multiply the numbers
            stors     f1,place1       # Save the result
            .
            .
            .
place1:     .space
```

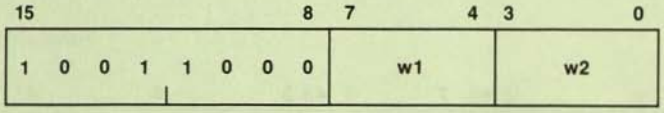
Syntax: **mulw** *w1,w2*

Description: Multiply the contents of general register w2 by general register w1 and and put the least-significant word of the product in w2. The operands are treated as signed integers. Overflow is set if the product cannot be represented in one word.

Operation:
w2 ← (w2) × (w1)
N ← 0
Z ← 0
V ← integer overflow
C ← 0

Traps: none

Format: Register



Example:

```
loadi    $999,r4     # Load 999 into r4
loadw    wdata,r2    # Get value at wdata
mulw     r2,r4       # Multiply the numbers
```

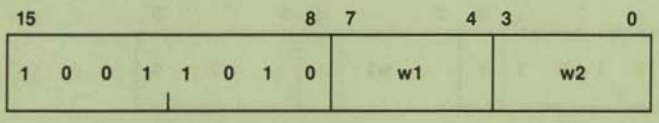
Syntax: **mulwu** *w1,w2*

Description: Multiply the contents of general register w2 by the contents of general register w1 and put the least-significant word of the product in w2. The operands are treated as unsigned integers. Overflow is set if the result cannot be represented in one word.

Operation:
w2 ← (w2) x (w1)
N ← 0
Z ← 0
V ← integer overflow
C ← 0

Traps: none

Format: Register



Example:

```
loadhu    hwval,r0    # Load the value at hwval
loadq     $33,r1      # Load 33 into r1
mulwu     r0,r1       # Multiply the numbers
```

Syntax: **mulwux** *w1,l2*

Description: Multiply the contents of general register w1 by the contents of general register w2 and put the product in longword register pair l2. The operands are treated as unsigned integers. Overflow is set if the product cannot be represented in one word.

Operation:
l2 ← (w2) × (w1)
N ← 0
Z ← 0
V ← result requires a longword
C ← 0

Traps: none

Format: Register

Example:

```
loadhu    uhwval,r3    # Load the value at uhwval
loadi     $0xff,r6     # Load ff hex into r6
mulwux    r3,r6        # Multiply the numbers
```

MULWX — Multiply Word-Extended — MULWX

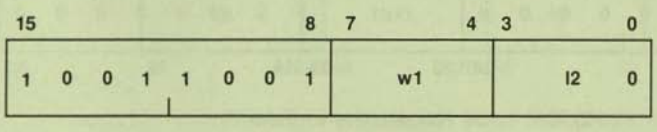
Syntax: **mulwx** *w1,l2*

Description: Multiply the contents of general register w1 by the contents of general register w2 and put the product in longword register pair l2. Overflow is set if the product cannot be represented in one word.

Operation:
l2 ← (w2) × (w1)
N ← 0
Z ← 0
V ← product requires a longword
C ← 0

Traps: none

Format: Register



Example:

```
loadhu    wxval,r0      # Load the value at wxval
loadi     $1023,r4      # Load 1023 into r4
mulwx     r0,r4         # Multiply the numbers
```

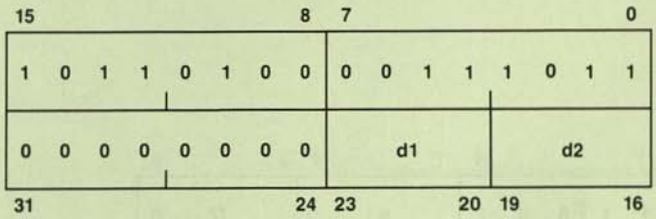
Syntax: **negd** *d1,d2*

Description: Negate the double-precision contents of floating register d1 and put the result in floating register d2. The sign of d1 is reversed so that ±0.0 and NaNs are handled properly.

Operation: d2 ← – (d1)

Traps: none

Format: Macro



Example:

```
negd    f0,f0    # Negate the register
```

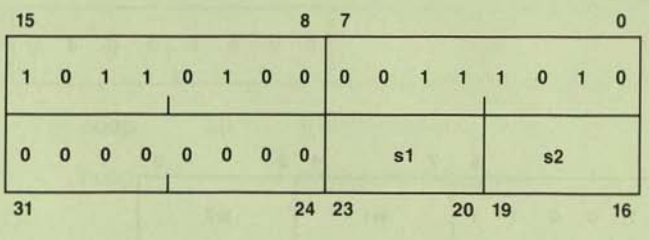
Syntax: **negs** *s1,s2*

Description: Negate the single-precision contents of floating register s1 and put the result in floating register s2. The sign of s1 is reversed so that ±0.0 and NaNs are handled properly.

Operation: s2 ← −(s1)

Traps: none

Format: Macro



Example:
```
negs    f1,f2    # Negate the value in f1, store in f2
```

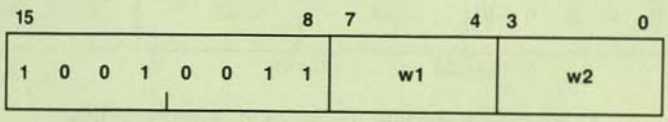
Syntax: **negw** *w1,w2*

Description: Two's complement the contents of general register w1 and put the result in general register w2. Overflow is set if w1 contains the largest negative number (-2^{31}). Carry is set if w1 does not equal 0.

Operation:
w2 ← −(w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

Traps: none

Format: Register



Example:
```
negw    r1,r1    # Two's complement (r1)
```

NOOP

No Operation

NOOP

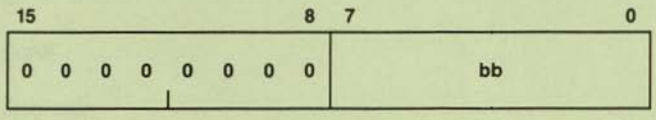
Syntax: **noop** *bb*

Description: No operation is performed. bb is ignored.

Operation: none

Traps: none

Format: Control



Examples:

```
noop    $0      # No operation
noop            # Operand is optional
```

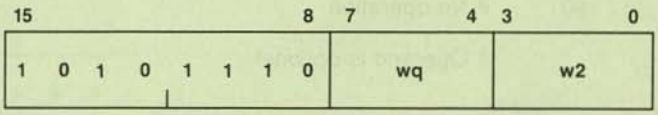
Syntax: **notq** *wq,w2*

Description: Zero fill the quick value wq on the left, take its one's complement and put the result in general register w2.

Operation:
w2 ← ~wq
N ← 1
Z ← 0
V ← 0
C ← 0

Traps: none

Format: Quick



Example:
```
notq    $4,r0    # Load -5 into r0
```

NOTW — Not Word — NOTW

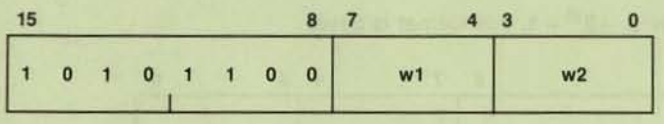
Syntax: **notw** *w1,w2*

Description: Take the one's complement of the contents of general register w1 and put the result in general register w2.

Operation:
w2 ← ~(w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Example:
```
notw    r1,r2    # One's complement r1, put in r2
```

Syntax: **ori** *wi,w2*

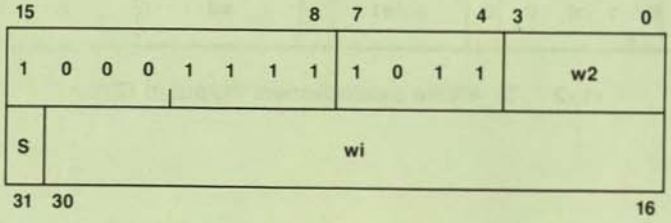
Description: Bitwise OR the contents of general register w2 with immediate value wi, and put the result in w2. A 16-bit immediate value is sign-extended first.

Operation:
w2 ← (w2) | wi
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

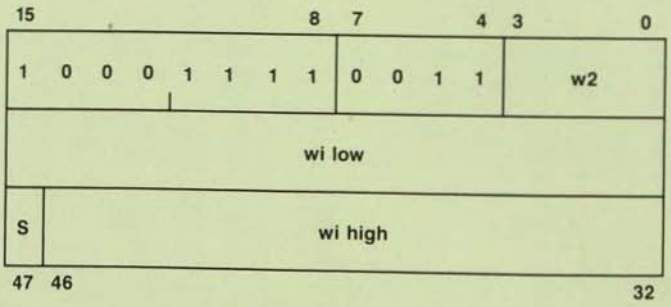
Traps: none

Formats: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{15} - 1$, this format is used:



Example: Assume r0 contains 0x00ff00ff.

```
ori     $0xff,r0        # Or 0xff with (r0)
```

The result put in r0 is 0x00ff00ff.

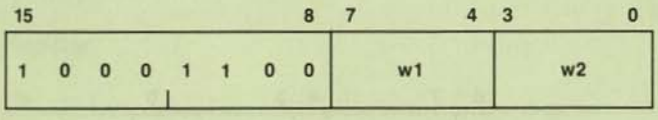
Syntax: **orw** *w1,w2*

Description: Bitwise OR the contents of general register w2 with the contents of general register w1 and put the result in w2.

Operation:
w2 ← (w2) | (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Example: Assume r0 contains 0x7770088f and r1 contains 0x001100ff.

```
orw     r1,r0     # Or (r0) with (r1)
```

The result put in r0 is 0x777108ff.

Syntax: **popw** *w1,w2*

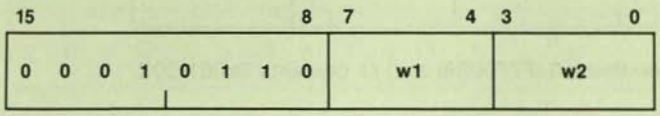
Description: Pop the word on the top of the stack into general register w2. General register w1 contains the stack address. The stack grows from high to low addresses.

On a data page fault, the contents of w1 will have been incremented. The page fault handler must check for this case and decrement the stack pointer by 4 before restarting the instruction.

Operation:
w1 ← (w1)+4
w2 ← ((w1)−4)

Traps:
Page fault
Read protect fault
Memory fault

Format: Register



Example: Assume sp = r15 = stack pointer.

```
popw    sp,r0    # Pop word into r0
```

PUSHW — Push Word — PUSHW

Syntax: **pushw** *w2,w1*

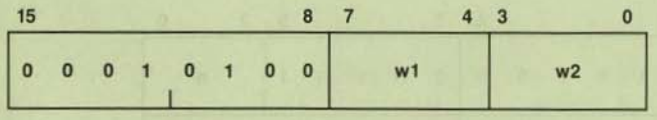
Description: Push the contents of general register w2 onto the stack. General register w1 contains the stack address. The stack grows from high to low addresses.

On a data page fault, the contents of w1 will have been decremented. The page fault handler must check for this case and increment the stack pointer by 4 before restarting the instruction.

Operation:
w1 ← (w1) – 4
(w1) ← (w2)

Traps:
Page fault
Write protect fault
Memory fault

Format: Register



Example: Assume sp = r15 = stack pointer.

```
pushw    r2,sp    # Push (r2) onto stack
```

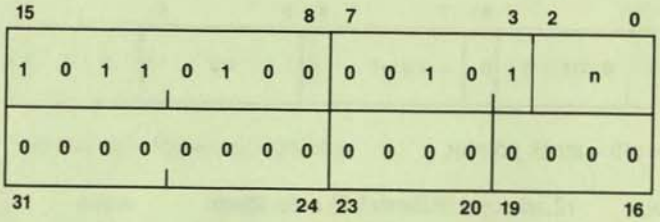
Syntax: **restd*n***

Description: This description represents the eight instructions **restd**0 through **restd**7. The **restd***n* instruction restores the double-precision, floating registers f*n* through f7 from the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting. A floating register containing a single-precision value will be restored properly by the appropriate **restd***n* instruction, but will not appear in IEEE single-precision format while in memory.

Operation: dn : d7 ← ((r15)) : ((r15) + 8 × [7 – n])
r15 ← (r15) + 8 × [8 – n]

Traps: Page fault
Read protect fault
Memory fault

Format: Macro



Example:

```
restd3      # Restore floating registers f3 : f7
restd7      # Restore floating register f7
```

Syntax: **restur** *w1*

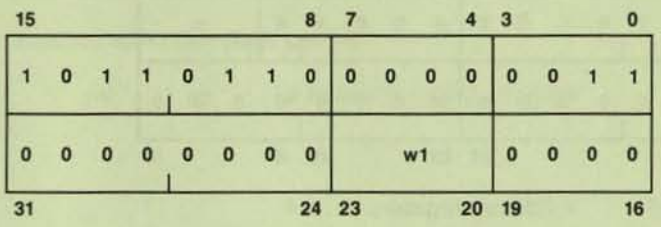
Description: Restore all the user registers. Restore the contents of all user registers r0 through r15 from supervisor memory addressed by supervisor general register w1. Register w1 may be the supervisor stack pointer, r15. On a data page fault, w1 is unchanged to permit restarting.

A privileged instruction trap occurs if this instruction is executed in user mode.

Operation: $r0 : r15_{usr} \leftarrow ((w1)) : ((w1) + 60)_{sup}$
$w1 \leftarrow (w1) + 64$

Traps: Privileged instruction
Page fault
Read protect fault
Memory fault

Format: Macro



Example:
```
restur    r1    # Restore user's registers
```

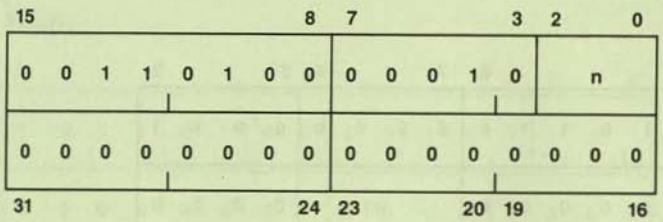
Syntax: **restw***n*

Description: This description represents the 13 instructions **restw**0 through **restw**12. The **restw***n* instruction restores the general registers r*n* through r14 from the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting.

Operation:
rn : r14 ← ((r15)) : ((r15) + 4 × (14 – n))
r15 ← (r15) + 4 × [15 – n]

Traps:
Page fault
Read protect fault
memory fault

Format: Macro



Example:
```
restw5      # Restore registers r5 : r14
```

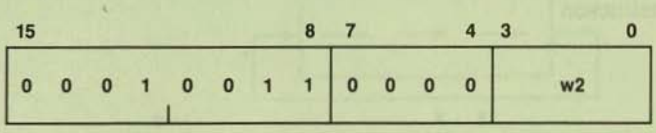
Syntax: **ret** *w2*

Description: Pop the word on top of the stack into the PC. General register w2 contains the stack address. This undoes the effect of the **call** instruction.

Operation:
PC ← ((w2))
w2 ← (w2) + 4

Traps:
Page fault
Read protect fault
Memory fault

Format: Register



Example: Assume sp = r15 = stack pointer.

```
ret     sp      # Return to calling program
```

Syntax: **reti** *w1*

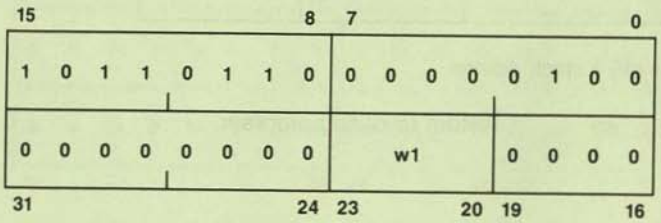
Description: Return from interrupt or trap. Restore the contents of the SSW, PSW, and PC from the supervisor stack addressed by the contents of general register w1. The supervisor must assure that the stack references do not cause page or protect faults.

A privileged instruction trap occurs if this instruction is executed in user mode.

Operation:
SSW ← ((w1))
PSW ← ((w1) + 4)
PC ← ((w1) + 8)
w1 ← (w1) + 12

Traps: Privileged instruction

Format: Macro

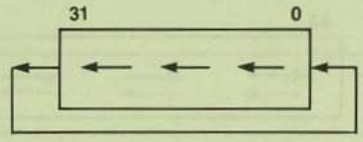


Example: Assume sp = r15 = stack pointer.

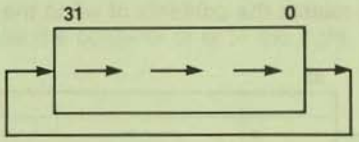
```
reti    sp      # Return from interrupt
```

Syntax: **roti** *wi,w2*

Description: Rotate the contents of general register w2 by the number of bits given in the 16-bit immediate value wi. A positive count rotates the contents of w2 to the left, moving bit 31 into bit 0:



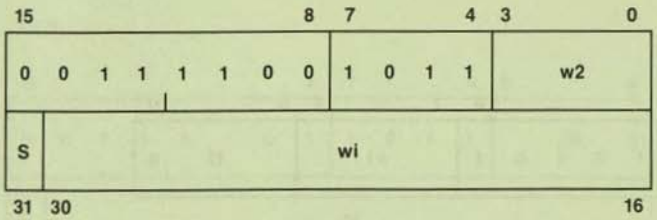
A negative count rotates to the right, moving bit 0 into bit 31:



Operation:
w2 ← (w2) ROT wi
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Immediate



Example: Assume r1 contains 0x3333ffff.

```
roti    $1,r1        # Rotate r1 left 1 place
```

The result put in r1 is 0x6667fffe.

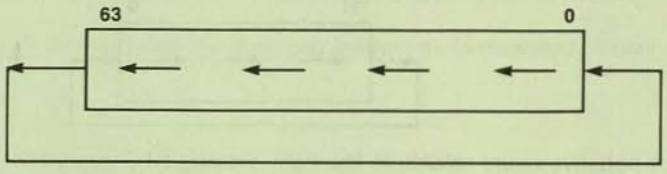
Assume r1 contains 0x3333ffff.

```
roti    $ – 1,r1     # Rotate r1 right 1 place
```

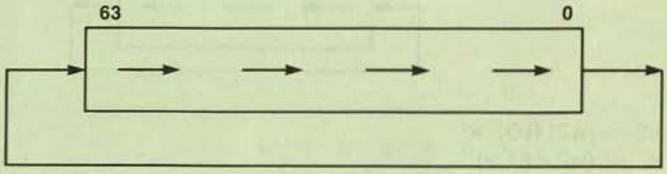
The result put in r1 is 0x9999ffff.

Syntax: **rotl** *w1,l2*

Description: Rotate the contents of longword register pair l2 by the number of bits given in general register w1. A positive count rotates the contents of l2 to the left, moving bit 63 into bit 0:



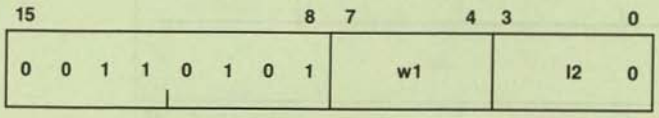
A negative count rotates the contents of w2 to the right, moving bit 0 into bit 63:



Operation:
l2 ← (l2) ROT (w1)
N ← (l2<63>)
Z ← (l2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Example: Assume r0 contains 2, r4,r5 contains 0x3333ffff 77770000.

```
rotl    r0,r4    # Rotate r4,r5 left 2 place
```

The result put in r4,r5 is 0xcccfffffbbb80000.

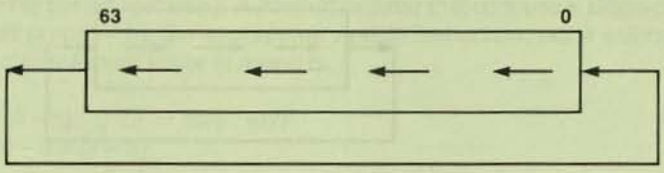
Assume r0 contains −2, r4,r5 contains 0x3333ffff 77770000.

```
rotl    r0,r4    # Rotate r4,r5 right 2 places
```

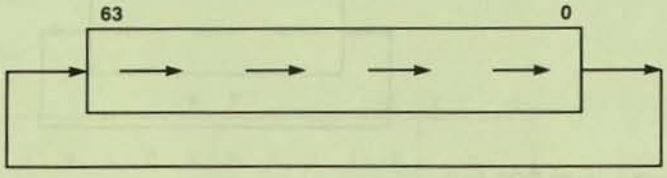
The result put in r4,r5 is 0x0cccfffffbbb8000.

Syntax: **rotli** *wi,l2*

Description: Rotate the contents of longword register pair l2 by the number of bits given in the 16-bit immediate value wi. A positive count rotates the contents of l2 to the left, moving bit 63 into bit 0:



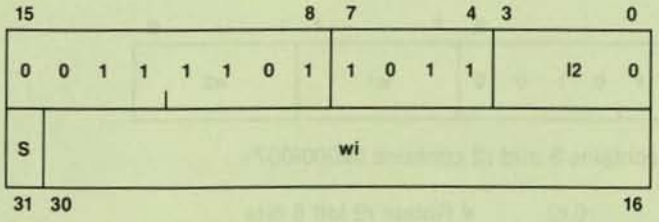
A negative count rotates the contents of l2 to the right, moving bit 0 into bit 63:



Operation:
l2 ← (l2) ROT wi
N ← (l2<63>)
Z ← (l2) = 0
V ← 0
C ← 0

Traps: none

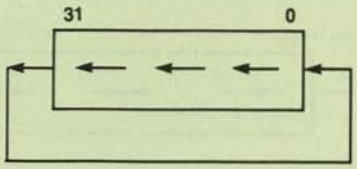
Formats:



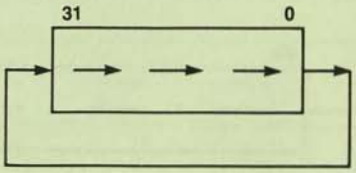
Example:
```
rotli    $8,r2    # Rotate left 1 byte
```

Syntax: **rotw** *w1,w2*

Description: Rotate the contents of general register w2 by the number of bits given in w1. A positive count rotates the contents of general register w2 to the left, moving bit 31 into bit 0:



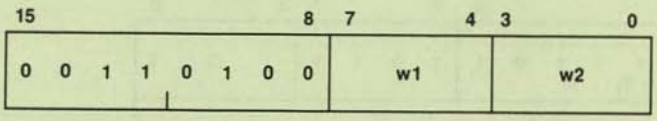
A negative count rotates right, moving bit 0 into bit 31:



Operation:
w2 ← (w2) ROT (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Example: Assume r0 contains 8 and r2 contains 0x000f0071.

```
rotw    r0,r2    # Rotate r2 left 8 bits
```

The result put in r2 is 0x0f007100.

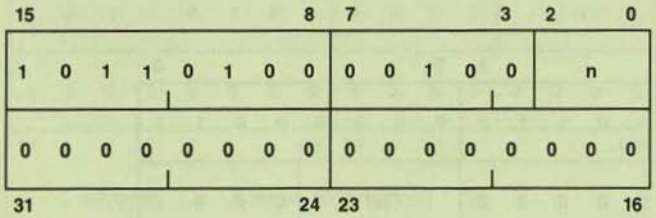
Syntax: **saved***n*

Description: This description represents the eight instructions **saved**0 through **saved**7. The **saved***n* instructions save the double-precision floating registers dn through d7 on the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting. A floating register that contains a single-precision value will be restored properly by the appropriate **restd***n* instruction, but it will not appear in IEEE single-precision format while in memory.

Operation: (r15) – 8 × [8 – n] : (r15) ← (dn) : (d7)
r15 ← (r15) – 8 × [8 – n]

Traps: Page fault
Write protect fault
Memory fault

Format: Macro



Example:
```
saved4      # Save floating registers f4 : f7
```

Syntax: **saveur** *w1*

Description: Save the contents of user general registers r0 through r15 in supervisor memory addressed by supervisor general register w1. Register w1 may be the supervisor stack pointer, r15. On a data page fault, w1 is unchanged to permit restarting.

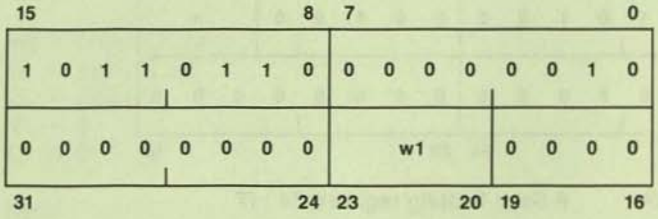
A privileged instruction trap occurs if this instruction is executed outside of supervisor mode.

Operation: $(w1) - 4 : (w1) - 64_{sup} \leftarrow (r15) : (r0)_{usr}$

$w1 \leftarrow (w1) - 64$

Traps: Privileged instruction
Page fault
Write protect fault
Memory fault

Format: Macro



Example: saveur r1 # Save user's registers r1 : r15

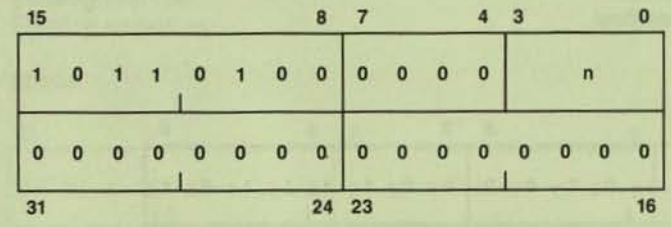
Syntax: **savew***n*

Description: This description represents the 13 instructions **savew**0, through **savew**12. The **savew***n* instruction saves the general registers rn through r14 on the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting.

Operation: (r15) – 4 × [15 – n] : (r15) ← (rn) : (r14)
r15 ← (r15) – 4 × [15 – n]

Traps: Page fault
Write protect fault
Memory fault

Format: Macro



Example:
```
savew6      # Save registers r6 : r14
```

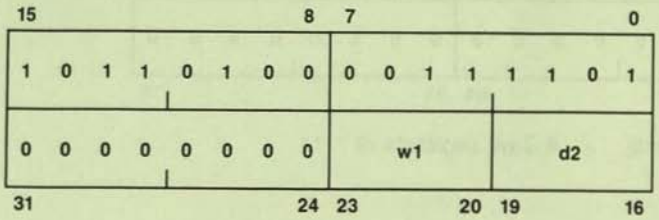
Syntax: **scalbd** *w1,d2*

Description: Multiply the double-precision contents of floating register d2 by two raised to the integer contents of general register w1 and put the result in d2. In the normal case, just the exponent is modified; a multiply operation is not performed. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
d2 ← (d2) x $2^{(w1)}$
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Macro



Example:
```
scalbd    r1,f0    # Scale the floating number
```

SCALBS — Scale By, Single Floating — SCALBS

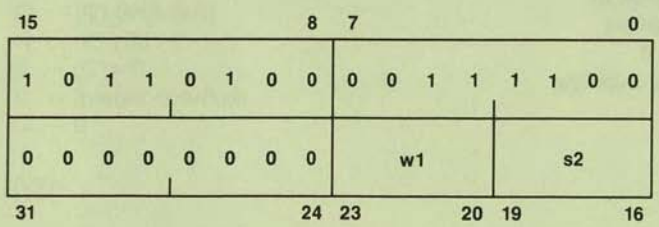
Syntax: **scalbs** *w1,s2*

Description: Multiply the contents of single-precision floating register s2 by two raised to the integer contents of general register w1 and put the result in s2. In the normal case, just the exponent is modified; a multiply operation is not performed. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
s2 ← (s2) x $2^{(w1)}$
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Macro

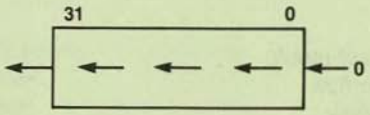


Example:
```
scalbs    r1,f0    # Scale the double floating value
```

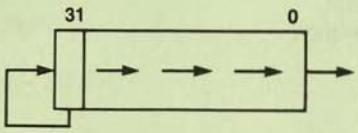
10

Syntax: **shai** *wi,w2*

Description: Shift arithmetically the contents of general register w2 by the number of bits given by the 16-bit immediate value wi. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of general register w2 left, bringing zeros into bit 0 and shifting the bit 31 out:



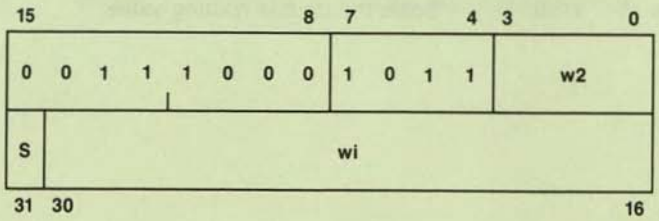
A negative count shifts right, bringing in copies of bit 31:



Operation:
w2 ← (w2) SHA wi
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← 0

Traps: none

Format: Immediate



Example: Assume r2 contains 0xffff0000.

```
shaw    $8,r2       # Shift r2 left 8 places
```

The result put in r2 is 0xff000000.

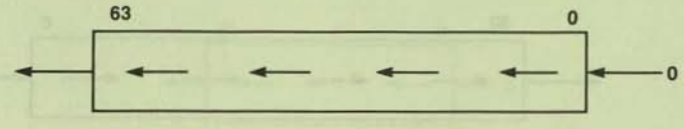
Assume r2 contains 0xffff0000.

```
shaw    $-8,r2      # Shift r2 right 8 places
```

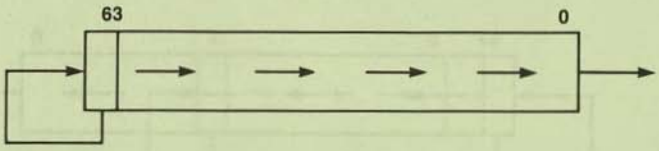
The result put in r2 is 0xffffff00.

Syntax: **shal** *w1,l2*

Description: Shift arithmetically the contents of longword register pair l2 by the number of bits given in w1. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



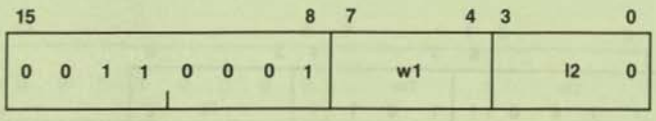
A negative count shifts right, bringing in copies of bit 63:



Operation:
l2 ← (l2) SHA (w1)
N ← (l2<63>)
Z ← (l2) = 0
V ← integer overflow
C ← 0

Traps: none

Format: Register



Example: Assume r1 contains 5, r6,r7 contains 0x1111ffff00ff00ff.

```
shal    r1,r6    # Shift r6,r7 left 5 places
```

The result put in r6,r7 is 0x223fffe01fe01fe0.

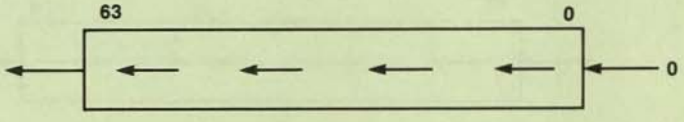
Assume r1 = – 5, r6,r7 = 0xffff111100ff00ff.

```
shal    r1,r6    # Shift r6,r7 right 5 places
```

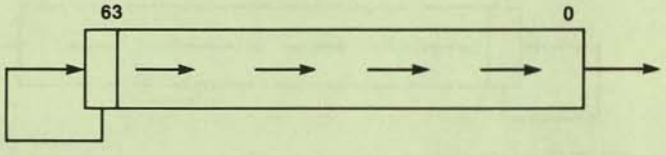
The result put in r6,r7 is 0xfffff8888 8807f807.

Syntax: **shali** *wi,l2*

Description: Shift arithmetically the contents of longword register pair l2 by the number of bits given in the 16-bit immediate value wi. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



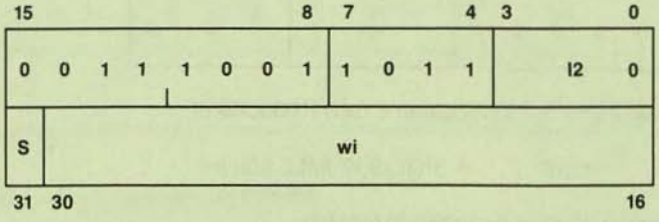
A negative count shifts right, bringing in copies of bit 63:



Operation:
l2 ← (l2) SHA wi
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← 0

Traps: none

Format: Immediate



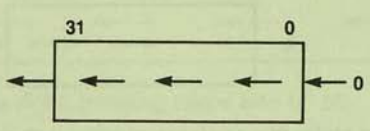
Example: Assume r8 contains 0x1110ffff.

```
shali    $-4,r8    # Shift r8 right 4 bits
```

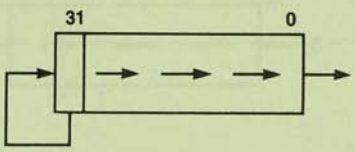
The result put in r8 is 0x01110ff.

Syntax: **shaw** *w1,w2*

Description: Shift arithmetically the contents of general register w2 by the number of bits given in general register w1. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of general register w2 to the left, bringing zeros into bit 0:



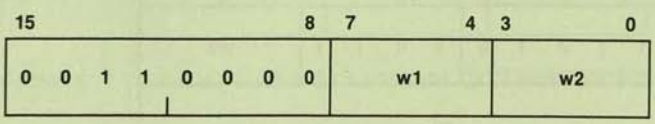
A negative count shifts right, bringing in copies of bit 31:



Operation:
w2 ← (w2) SHA (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← 0

Traps: none

Format: Register



Example: Assume r10 contains 8, r2 contains 0xffff0000.

```
shaw    r10,r2    # Shift r2 left 8 places
```

The result put in r2 is 0xff000000.

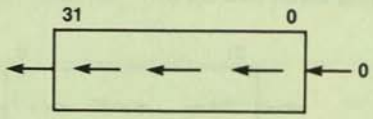
Assume r10 contains −8, r2 contains 0xffff0000.

```
shaw    r10,r2    # Shift r2 right 8 places
```

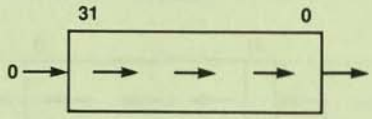
The result put in r2 is 0xffffff00.

Syntax: **shli** *wi,w2*

Description: Shift logically the contents of general register w2 by the number of bits given in the 16-bit immediate value wi. A positive count shifts the contents of w2 out the left; bringing zeros into bit 0:



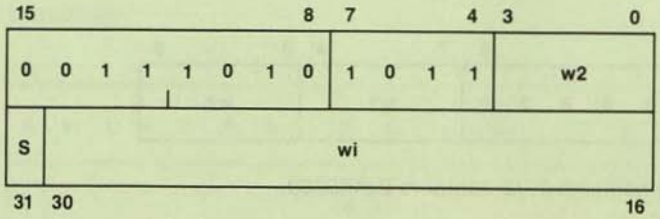
A negative count shifts the contents of w2 out the right; bringing zeros into bit 31.



Operation:
w2 ← (w2) SHL wi
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Immediate



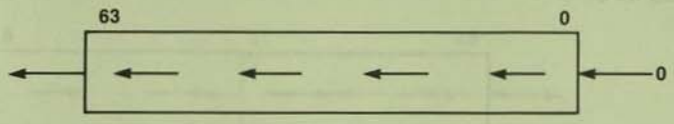
Example: Assume r4 contains 0xffff0000.

```
shli    $-8,r4      # Shift r4 right 8 places
```

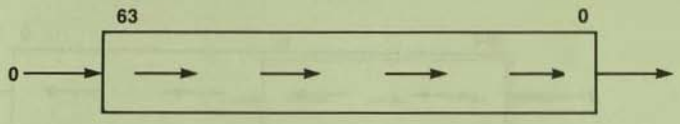
The result put in r4 is 0x00ffff00.

Syntax: **shll** *w1,l2*

Description: Shift logically the contents of longword register pair l2 by the number of bits given in general register w1. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



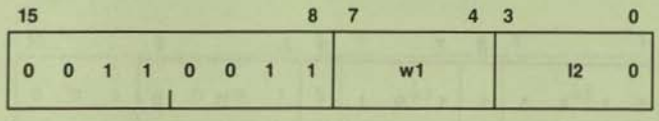
A negative count shifts right, bringing zeros into bit 63:



Operation:
l2 ← (l2) SHL (w1)
N ← (l2<63>)
Z ← (l2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Examples: Assume r1 contains 16 and r2,r3 contains 0x123456780000ffff.

```
shll    r1,r2    # Shift (r2,r3) by (r1)
```

The result put in r2,r3 is 0x56780000ffff0000.

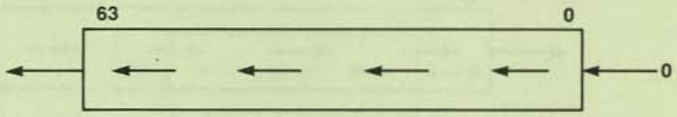
Assume r1 contains – 16 and r2,r3 contains 0x123456780000ffff.

```
shll    r1,r2    # Shift (r2,r3) by (r1)
```

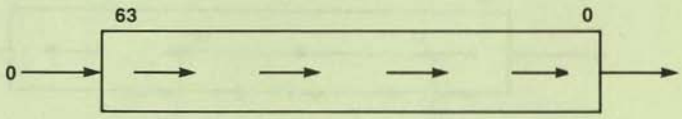
The result put in r2,r3 is 0x0000000012345678.

Syntax: **shlli** *wi,l2*

Description: Shift logically the contents of longword register pair l2 by the number of bits given in the 16-bit immediate value wi. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



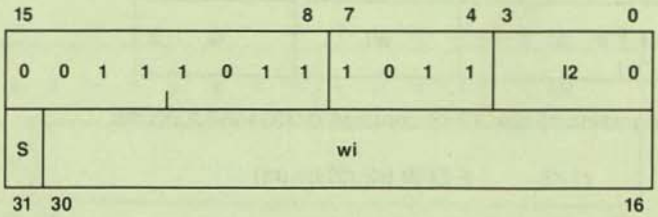
A negative count shifts right, bringing zeros into bit 63:



Operation:
l2 ← (l2) SHL wi
N ← (l2<63>)
V ← (l2) = 0
Z ← 0
C ← 0

Traps: none

Format: Immediate



Example: Assume r2,r3 contains 0x7777ffff7777ffff.

```
sh11i     $8,r2     # Shift (r2,r3) left 8 places
```

The result put in r2,r3 is 0x77ffff7777ffff00.

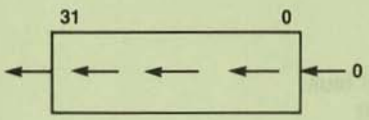
Assume r2,r3 contains 0x7777ffff7777ffff.

```
sh11i     $-8,r2    # Shift (r2,r3) right 8
```

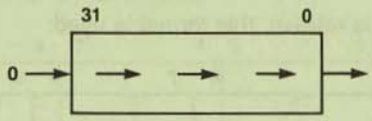
The result put in r2,r3 is 0x007777ffff7777ff.

Syntax: **shlw** *w1,w2*

Description: Shift logically the contents of general register w2 by the number of bits given in general register w1. A positive count shifts the contents of general register w2 to the left, bringing zeros into bit 0:



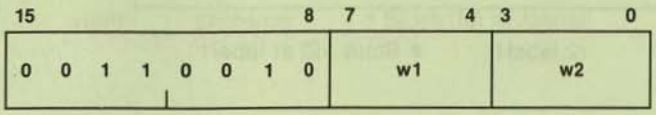
A negative count shifts right, bringing zeros into bit 31:



Operation:
w2 ← (w2) SHL (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Examples: Assume r1 contains 4 and r0 contains 0xff.

```
shlw    r1,r0    # Shift (r0) by (r1)
```

The result put in r0 is 0x000000ff0.

Assume r1 contains −4 and r0 contains 0xff.

```
shlw    r1,r0    # Shift (r0) by (r1)
```

The result put in r0 is 0x0000000f.

Syntax: **storb** *w2,ba*

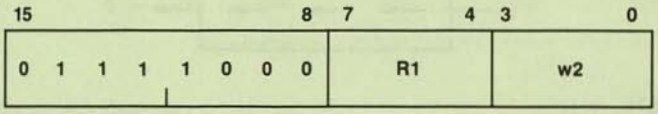
Description: Store the least-significant byte of the contents of general register w2 into memory address ba.

Operation: ba ← (w2)

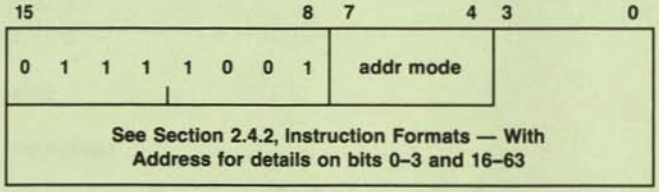
Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
storb    r2,label1    # Store (r2) at label1
```

Syntax: **stord** *d2,da*

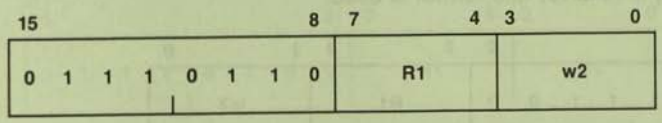
Description: Store the double-precision contents of floating register d2 into memory addresses da.

Operation: da ← (d2)

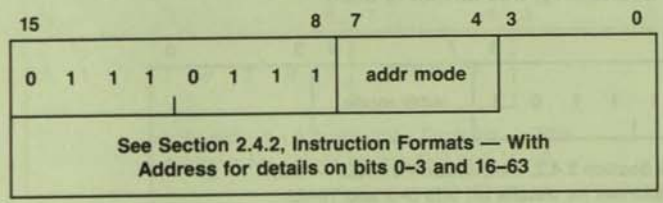
Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
stord    f0,fpsave    # Store (f0) at fpsave
```

Syntax: **storh** *w2,ha*

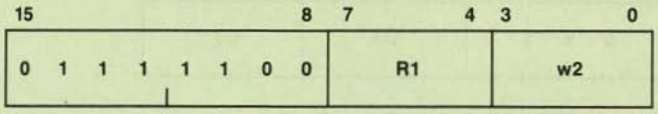
Description: Store the least-significant halfword of the contents of general register w2 into memory address ha.

Operation: ha ← (w2)

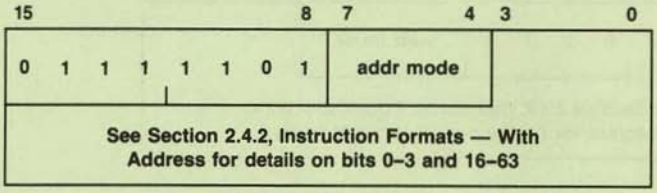
Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
storh    r12,hwsav1    # Store (r12) at hwsav1
```

Syntax: **stors** *s2,sa*

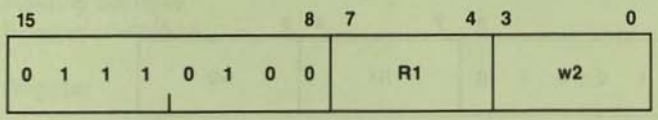
Description: Store the single-precision contents of floating register s2 into memory address sa.

Operation: sa ← (s2)

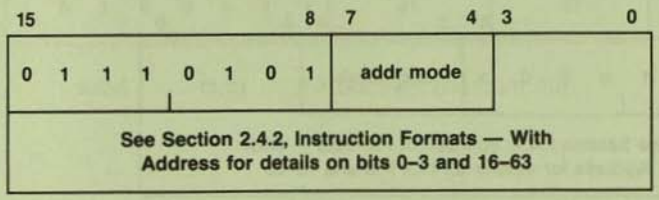
Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:
```
stors    f0,sfbuff    # Store (f0) at sfbuff
```

Syntax: **storw** *w2,wa*

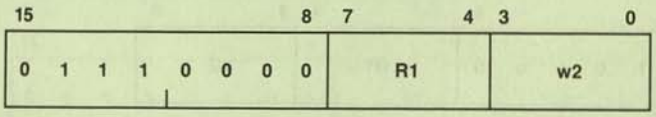
Description: Store the contents of general register w2 into memory address wa.

Operation: wa ← (w2)

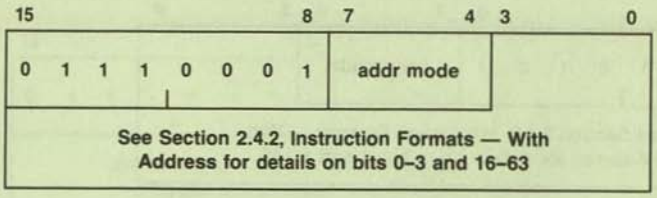
Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
storw    r0,waddr    # Store (r0) at waddr
```

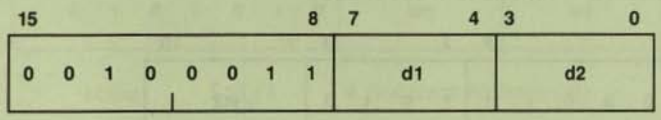
Syntax: **subd** *d1,d2*

Description: Subtract the double-precision contents of floating register d1 from the double-precision contents of floating register d2 and put the result in d2.

Operation:
d2 ← (d2) – (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register



Example:
```
subd    f1,f0    # Subtract (f1) from (f0)
```

Syntax: **subi** *wi,w2*

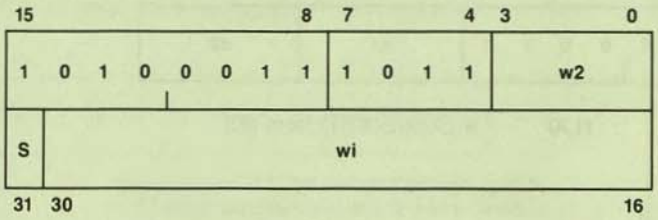
Description: Subtract the immediate value wi from the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the input operands (which are treated as signed integers) have different signs and the sign of the result has the same sign as wi.

Operation:
w2 ← (w2) – wi
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

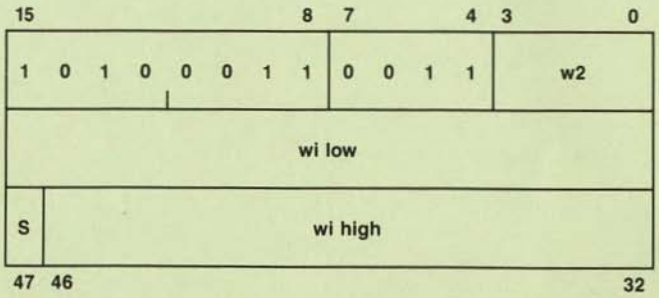
Traps: none

Format: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{15} - 1$, this format is used:



Example:
```
        subi    $0x18,r2        # Subtract 18 hex from (r2)
```

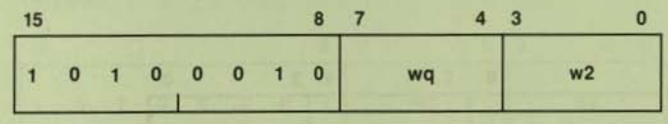
Syntax: **subq** *wq,w2*

Description: Subtract the quick value wq from the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the input operands (which are treated as signed integers) have different signs and the result is positive.

Operation:
w2 ← (w2) – wq
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

Traps: none

Format: Quick



Example:
```
subq    $10,r3      # Subtract 10 from (r3)
```

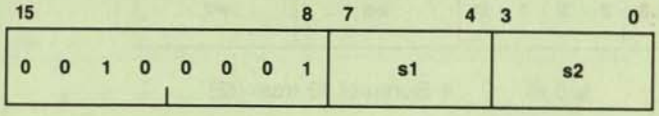
Syntax: **subs** *s1,s2*

Description: Subtract the single-precision contents of floating register s1 from the single-precision contents of floating register s2 and put the result in d2.

Operation:
s2 ← (s2) – (s1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register



Example:
```
subs    f1,f2    # Subtract (f1) from (f2)
```

SUBW — Subtract Word — SUBW

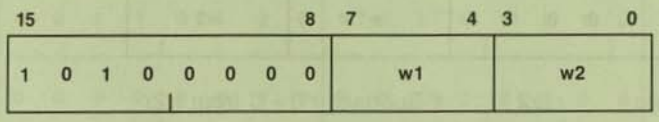
Syntax: **subw** *w1,w2*

Description: Subtract the contents of general register w1 from the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the operands (which are treated as signed integers) have different signs and the sign of the result is the same as the subtrahend (w1).

Operation:
w2 ← (w2) – (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

Traps: none

Format: Register



Example:
```
subw .    r0,r2      # Subtract (r0) from (r2)
```

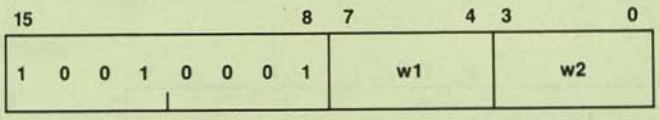
Syntax: **subwc** *w1,w2*

Description: Subtract the contents of general register w1 and the carry condition code from the contents of general register w2 and put the result in w2.

Operation:
w2 ← (w2) – (w1) – C
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

Traps: none

Format: Register



Example:
```
subwc    r1,r2    # Subtract (r1) + C from (r2)
```

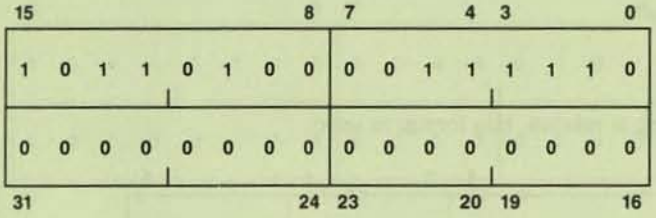
Syntax: **trapf***n*

Description: Causes an illegal instruction trap if a floating unordered condition exists. The IEEE draft standard specifies that the six predicates =, ≠, >, ≥, <, and ≤ shall cause floating invalid exceptions on unordered comparisons. The **trapf***n* instruction, put before a branch instruction, supports the IEEE predicates. The supervisor trap handler must interpret an illegal instruction trap from a **trapf***n* instruction as if it were a floating invalid operation trap.

Operation: if PSW <ZN> indicates unordered,
then illegal instruction trap

Traps: Illegal instruction

Format: Macro



Example: The following sequence branches to **addr** if the IEEE ≤ predicate is satisfied.

```
cmps    f3,f5    # Compare single floating values
trapfn           # Checks for floating unordered
bclt    addr     # branches
```

Syntax: **tsts** *wa,w2*

Description: Test and set a software lock. Load the contents of memory address wa into general register w2 and set bit 31 in wa (the lock). The operation is indivisible and can be used in a multiprocessor configuration. The lock has been acquired if (w2<31>) = 0 after the instruction has executed.

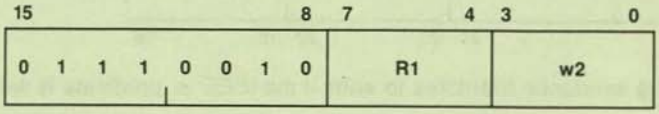
This instruction may not refer to Boot ROM space or I/O space (System Tag 4 or 5).

Operation: w2 ← (wa)
(wa<31>) ← 1

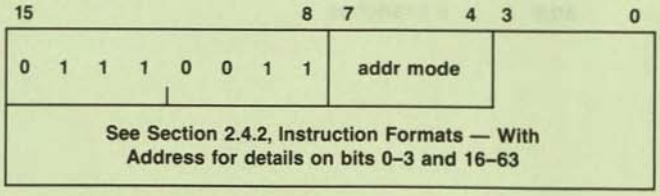
Traps: Page fault
Read protect fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



Example:

```
tsts    addr,r2    # Set software lock
```

Syntax: **wait**

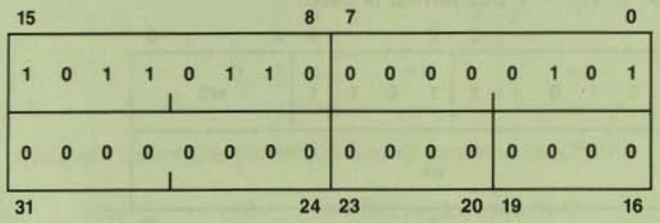
Description: Wait for an interrupt. When an enabled interrupt occurs, the instruction terminates and the interrupt is taken. The interrupt routine may then decide to whether or not to continue the interrupted instruction stream at the instruction following the **wait** instruction.

A privileged instruction trap occurs if this instruction is executed in user mode.

Operation:
```
while no interrupt pending
do nothing
```

Traps: Privileged instruction

Format: Macro



Example:
```
        wait        # Wait for interrupt
```

Syntax: **xori** *wi,w2*

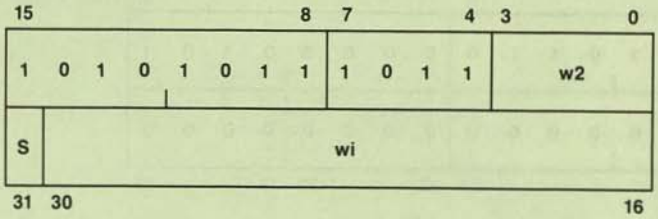
Description: Bitwise exclusive-OR the contents of general register w2 with the immediate value wi and put the result in w2.

Operation:
w2 ← (w2) ⊕ wi
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

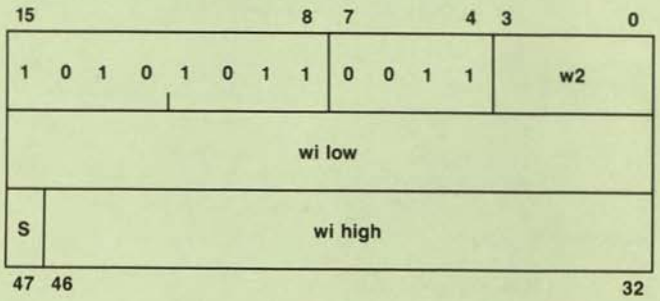
Traps: none

Formats: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{15} - 1$, this format is used:



Examples: Assume r0 contains 0xff00.

```
xori    $0xa07f601f,r0      # XOR 0x7f601f with (r0)
```

The result put in r0 is 0xa07f9f1f.

Assume r2 contains 0xc0ffee12.

```
xori    $0xfe,r2            # XOR 0xfe with (r2)
```

The result put in r2 is 0xc0ff10.

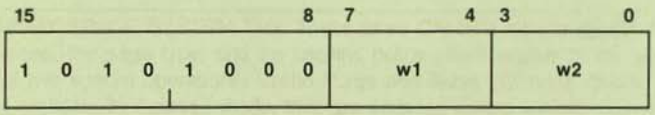
Syntax: **xorw** *w1,w2*

Description: Bitwise exclusive-OR the contents of general register w1 with the contents of general register w2 and put the result in w2.

Operation:
w2 ← (w2) ⊕ (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register



Example: Assume r1 contains 0xffff0000 and r2 contains 0x00ff00ff.

```
xorw    r1,r2    # XOR regs, result in r2
```

The result put in r2 is 0xff0000ff.

Appendix A
CLIPPER Module Interface Signal Descriptions

Signal	Description
AD <31:0>	ADDRESS/DATA. This is a positive logic (HIGH=logic 1) multiplexed address and data bus which is tied to the CAMMUs.
$\overline{DIR}$	DIRECTION CONTROL. This control signal can be used to control the drive direction of tranceivers interfacing AD <31:0> to the CLIPPER Bus. A master CAMMU asserts $\overline{DIR}$ during an entire write operation and during the first two cycles of a read operation. A slave D-CAMMU asserts this signal when transferring data during an I/O read; a slave I-CAMMU asserts this signal when transferring data to a companion D-CAMMU. Drive direction is from CAMMU to the CLIPPER Bus when $\overline{DIR}$ is LOW.
TG <2:0>	MEMORY SPACE SYSTEM TAG. These three CAMMU signals identify the address space being accessed, the page type, and the caching policy which applies to the accessed page. In addition, they signal two special operations, Cache Purge and Slave I/O mode. System tags are derived from one of three sources. In mapped mode, they are selected during address translation from either page tables in main memory, the TLB or the HTLB. In unmapped mode, TG<2> is zero and TG<1:0> is selected by the UST bits in the CAMMU Control Register. TG<2:0> tag encoding is as follows:

TG2	TG1	TG0	Description
0	0	0	main memory/private cacheable/write-through
0	0	1	main memory/shared cacheable/write-through
0	1	0	main memory/private cacheable/copy-back
0	1	1	main memory/noncacheable
1	0	0	I/O space/noncacheable
1	0	1	boot space/noncacheable
1	1	0	cache purge
1	1	1	slave I/O mode main memory/noncacheable

Note: The slave CAMMU can continually monitor the Memory Space System Tag and check for cache/main memory data consistency when the Bus Watch modes are enabled. The Bus Watch modes, when enabled, are invoked during the following CLIPPER Bus operations:

(1) I/O writes to shared or private pages.

(2) CPU writes to shared pages.

(3) I/O reads from private copy-back pages.

Frequently systems require the transfer of data between main memory and an I/O device. This type of data transfer is normally implemented by a CPU as a read operation into a CPU register, followed by a CPU write operation to the I/O device. This type of operation is accelerated by the CLIPPER Module through the use of the slave I/O mode identified by the Memory Space System Tag. The slave I/O mode allows an I/O device to capture data being read by the CPU during the read portion of the operation. The I/O device must be able to interpret TG=7 as slave I/O mode, then read the transferred data as it is being read by the CPU. The data read by the CPU is discarded.

CT <5:0>

CYCLE TYPE. These six CAMMU signals indicate the type of CLIPPER Bus operation in progress. CT<5:0> signal encoding is as follows:

Signal	State	Operation
CT5	0	CPU master.
	1	I/O master.
CT4	0	Write operation.
	1	Read operation.
CT<3:2>	00	Word/halfword/byte transfer.
	01	Quadword transfer.
	10	Reserved.
	11	Global CAMMU write.
CT<1:0>	00	Whole word transfer; AD<1:0> must be 0.
	01	Reserved
	10	Byte transfer; AD<1:0> define the byte position.
	11	Halfword transfer; AD<1:0> defines the halfword position; AD<1:0> must be 0.

Notes:

(a) CT<1:0> have meaning only for single word transfers, with AD<1:0> pointing to transferred word/bytes.

(b) In halfword/byte mode, the data must be written in the location specified by AD<1:0>.

(c) During a quadword transfer, the master must assert AD<3:0> all 0 to point to a quadword boundary.

CBSYi, CBSYd

CACHE BUSY (I-CAMMU, D-CAMMU). A CAMMU asserts CBSY to indicate execution of internal operations associated with Bus Watch operations. Main memory data must not be asserted on the CLIPPER Bus while CBSYi or CBSYd is asserted. If a CAMMU asserts RDYo while asserting CBSY, indicating that it is asserting more recent cache data on the CLIPPER Bus, main memory must abort the data transfer operation.

$\overline{\text{LOCK}}$

BUS LOCK. $\overline{\text{LOCK}}$ is asserted by a bus master when it requires the CLIPPER Bus for more than one operation. $\overline{\text{LOCK}}$ is asserted by the CAMMUs during the following operations:

(1) DTU Page Table Directory and Page Table accesses.

(2) DTU R or D bit modifications in Page Tables.

(3) Read-modify-write (test and set) operations.

(4) Cache line replace and fetch on cache miss (quadword write followed by quadword read).

$\overline{TR}$	TRANSFER REQUEST. $\overline{TR}$ is asserted by a bus master to indicate that a CLIPPER Bus operation is in progress.
RDYi	READY INPUT. RDYi is tied to both CAMMUs. During read operations, the slave with the addressed data asserts RDYi to indicate that it has asserted the data on the AD bus. During single word, byte or halfword write operations, the slave asserts RDYi to indicate that it has latched (read) the data. During quadword write operations, the slave asserts RDYi to indicate that it has latched the data word currently on the AD bus, and is ready to latch the next word.
RDYo	READY OUTPUT. RDYo is asserted by the D-CAMMU during Watch I/O Reads operation to indicate to the the I/O device that it has asserted data on the AD bus for reading. This occurs when the data location being accessed in main memory is cached, and the cache data is more "recent" than the main memory data. RDYo can be tied to $\overline{RDY}$ on the CLIPPER Bus.
RDYoi	READY OUTPUT. RDYoi is asserted by the I-CAMMU in response to an access through the companion D-CAMMU. RDYoi is not interfaced to the CLIPPER Bus.
BRi,BRd	BUS REQUEST (I-CAMMU, D-CAMMU). These signals are asserted by the respective CAMMUs to obtain control of the CLIPPER Bus.
BGi,BGd	BUS GRANT (I-CAMMU, D-CAMMU). Bus Grant is asserted by the CLIPPER Bus arbitration logic in response to a Bus Request by a CAMMU, and indicates that the requesting CAMMU has control of the bus.
$\overline{MSBE}$/ $\overline{RETRY}$	MEMORY SINGLE BIT ERROR/RETRY. The main memory interface asserts $\overline{MSBE/RETRY}$ when it detects a corrected single-bit error in main memory during a read operation. $\overline{MSBE/RETRY}$ is tied to both CAMMUs, and is sampled by the CAMMUs when RDYi is sampled. A master CAMMU issues a trap to the CPU when it detects $\overline{MSBE/RETRY}$ asserted. The main memory interface must not assert an interrupt when it detects a single-bit data error. The $\overline{MSBE}$ signal must be presented to the CAMMU by the memory interface along with (during the same BCLK as) RDYi and the data to indicate a corrected single-bit error. The $\overline{MSBE/RETRY}$ signal is also used to abort and retry CLIPPER bus operations. If the signal is asserted during access of I/O space (TG=4) while RDYi is inactive, the current bus operation is aborted by the master CAMMU with no trap assertion to the CPU.
$\overline{MMBE}$	MEMORY MULTIPLE BIT ERROR. The main memory interface asserts $\overline{MMBE}$ when it detects an uncorrectable multiple-bit error in main memory during a read operation. $\overline{MMBE}$ is tied to both CAMMUs, and is sampled by the CAMMUs when RDYi is sampled. A master CAMMU issues a trap to the CPU when it detects $\overline{MMBE}$ asserted. The main memory interface must not assert an interrupt when it detects a multiple-bit data error.

$\overline{\text{BERR}}$	BUS ERROR. $\overline{\text{BERR}}$ should be asserted by user-designed logic to indicate a CLIPPER Bus error condition such as a bus timeout. $\overline{\text{BERR}}$ is tied to both CAMMUs. Upon assertion of $\overline{\text{BERR}}$, the master CAMMU terminates the bus operation and indicates to the CPU that the operation is completed. (If the bus error occurs during a read operation, the data asserted on the AD bus at the time $\overline{\text{BERR}}$ is asserted is transferred by the CAMMU to the CPU). The CAMMU does not issue a trap in response to a bus error; the bus error logic should assert an interrupt
$\overline{\text{IVEC}}$ <7:0>	INTERRUPT VECTORS. This is an inverted logic (LOW=logic 1) bus, tied directly to the CPU. It transfers interrupt vector numbers associated with interrupt requests.
$\overline{\text{IRQ}}$	INTERRUPT REQUEST. This signal, tied directly to the CPU, is asserted by system devices for interrupt service requests. Once asserted, $\overline{\text{IRQ}}$ must remain asserted until $\overline{\text{IACK}}$ is asserted by the CPU. An interrupt level and number must be asserted on $\overline{\text{IVEC}}$<7:0> with each interrupt request. $\overline{\text{IRQ}}$ is maskable.
$\overline{\text{IACK}}$	INTERRUPT ACKNOWLEDGE. $\overline{\text{IACK}}$ is asserted by the CPU in response to an interrupt request ($\overline{\text{IRQ}}$) to acknowledge that servicing of the interrupt is in progress.
$\overline{\text{NMI}}$	NON-MASKABLE INTERRUPT. This signal, tied directly to the CPU, is asserted by system devices for non-maskable interrupt service requests. Once asserted, $\overline{\text{NMI}}$ must remain asserted until $\overline{\text{NMIACK}}$ is asserted by the CPU.
$\overline{\text{NMIACK}}$	NON-MASKABLE INTERRUPT ACKNOWLEDGE. This signal is asserted by the CPU in response to an $\overline{\text{NMI}}$ request to acknowledge that servicing of the interrupt is in progress.
RATE	BCLK RATE CONTROL. This input to the CLIPPER Module controls the CLIPPER Bus BCLK frequency. When RATE is tied to GND, BCLK frequency is 1/2 MCLK frequency. When RATE is tied to V_{DD}. BCLK frequency is 1/4 MCLK frequency. If the standard 66.7 MHz oscillator is used in the CLIPPER Module, BCLK frequency is 16.7 MHz if RATE is tied to GND, and 8.3 MHz if RATE is tied to V_{DD}.
BCLK	BUS CLOCK. BCLK clocks all devices on the CLIPPER Bus. All signals must be clocked onto the CLIPPER Bus on the rising edge of BCLK; all signals must be latched/sampled from the CLIPPER Bus on the rising edge of BCLK. The propagation delay of signals asserted on the bus must be less than one BCLK period and more than the BCLK skew between devices in order to ensure proper operation of the synchronous CLIPPER Bus.
$\overline{\text{RESET}}$	This is the master reset signal which is asserted by system logic to reset the CLIPPER Module and other devices on the CLIPPER Bus. Upon release of $\overline{\text{RESET}}$, the CPU begins instruction fetching at Boot space address 0.

$\overline{\text{URF}}$ UNRECOVERABLE FAULT. This signal is asserted by the CPU to indicate that it has stopped program execution as a result of an unrecoverable fault condition. An unrecoverable fault occurs when the CPU encounters an error condition during execution of on-chip diagnostic routines, or when a trap occurs during an INTRAP or **reti** execution sequence.

$\overline{\text{URDIAG}}$ APPLY DIAGNOSTICS. This input to the CPU is asserted to force the CPU to execute on-chip diagnostic routines resulting in the following:

(1) The diagnostics detected no error conditions. The CPU begins program execution at Boot ROM space address 0 (supervisor virtual address 6000 hex).
(2) The diagnostics detected an error condition. The CPU asserts $\overline{\text{URF}}$ and stops execution.

$\overline{\text{RESET}}$ must be asserted when $\overline{\text{URDIAG}}$ is asserted.

OSC OSCILLATOR INPUT. This input signal is used by the CLIPPER Clock Control Unit to derive MCLK and BCLK. MCLK is the CLIPPER internal clock; BCLK is the CLIPPER Bus clock.

Appendix B
Instruction Set Summary

This appendix summarizes the instruction set. For a detailed description of each instruction, see Chapter 10.

TABLE B.1 Instruction Set By Function

LOAD/STORE INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Load Address	loada	*ba,w2*	62,63	Address	1-4	w2 ← ba	....	
Load Byte	loadb	*ba,w2*	68,69	Address	1-4	w2 ← (ba)	....	P,R,M,
Load Byte Unsigned	loadbu	*ba,w2*	6a,6b	Address	1-4	w2 ← (ba)	....	P,R,M,
Load Double Floating	loadd	*da,d2*	66,67	Address	1-4	d2 ← (da)	....	P,R,M,
Load Floating Status	loadfs	*w1,d2*	b4 3f	Macro	2	w1 ← (FP PC), d2 ← (FP dest)	....	
Load Halfword	loadh	*ha,w2*	6c,6d	Address	1-4	w2 ← (ha)	....	P,R,M,
Load Halfword Unsigned	loadhu	*ha,w2*	6e,6f	Address	1-4	w2 ← (ha)	....	P,R,M,
Load Immediate	loadi	*wi,w2*	87	Immediate	2,3	w2 ← wi	..... 00**	
Load Quick	loadq	*wq,w2*	86	Quick	1	w2 ← wq	..... 00*0	
Load Single Floating	loads	*sa,s2*	64,65	Address	1-4	s2 ← (sa)	....	P,W,M,
Load Word	loadw	*wa,w2*	60,61	Address	1-4	w2 ← (wa)	....	P,R,M,
Store Byte	storb	*w2,ba*	78,79	Address	1-4	ba ← (w2)	....	P,W,M,
Store Double Floating	stord	*d2,da*	76,77	Address	1-4	da ← (d2)	....	P,W,M,
Store Halfword	storh	*w2,ha*	7c,7d	Address	1-4	ha ← (w2)	....	P,W,M,
Store Single Floating	stors	*s2,sa*	74,75	Address	1-4	sa ← (s2)	....	P,W,M,
Store Word	storw	*w2,wa*	70,71	Address	1-4	wa ← (w2)	....	P,W,M,

DATA MOVEMENT INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Move Double Floating	movd	*d1,d2*	26	Register	1	d2 ← (d1)	....	
Move Double Floating to Longword	movdl	*d1,l2*	2e	Register	1	l2 ← (d1)	....	
Move Longword to Double Floating	movld	*l1,d2*	2f	Register	1	d2 ← (l1)	....	
Move Processor Register to Word	movpw	*p1,w2*	11	Register	1	w2 ← (p1)	....	
Move Single Floating	movs	*s1,s2*	24	Register	1	s2 ← (s1)	....	
Move Supervisor to User (privilege)	movsu	*w1,w2*	b6 01	Macro	2	w2 ← (w1)	..... 00**	S
Move Single Floating to Word	movsw	*s1,w2*	2c	Register	1	w2 ← (s1)	....	
Move User to Supervisor (privilege)	movus	*w1,w2*	b6 00	Macro	2	w2 ← (w1)	..... 00**	S
Move Word	movw	*w1,w2*	84	Register	1	w2 ← (w1)	..... 00**	
Move Word to Processor Register	movwp	*w2,p1*	10	Register	1	p1 ← (w2)	..... ****	
Move Word to Single Floating	movws	*w1,s2*	2d	Register	1	s2 ← (w1)	....	

Legend: PSW Flags Field
- . = Flag not affected by instruction
- * = Flag set according to operation
- 0 = Flag set to 0
- 1 = Flag set to 1

Traps Field
- D = Divide-by-zero
- I = Illegal instruction
- M = Memory fault
- P = Page fault
- R = Read protect fault
- S = Supervisor only (privileged) instruction
- W = Write protect fault

TABLE B.1 Instruction Set By Function (Continued)

ARITHMETIC INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	PSW Flags CVZN	Traps
Add Double Floating	addd	*d1,d2*	22	Register	1	d2 ← (d2) + (d1)	**.**	....	
Add Immediate	addi	*wi,w2*	83	Immediate	2,3	w2 ← (w2) + wi	.....	****	
Add Quick	addq	*wq,w2*	82	Quick	1	w2 ← (w2) + wq	.....	****	
Add Single Floating	adds	s1,s2	20	Register	1	s2 ← (s2) + (s1)	** **	....	
Add Word	addw	*w1,w2*	80	Register	1	w2 ← (w2) + (w1)	.....	****	
Add Word with Carry	addwc	*w1,w2*	90	Register	1	w2 ← (w2) + (w1) + C	.....	****	
Subtract Double Floating	subd	*d1,d2*	23	Register	1	d2 ← (d2) – (d1)	**.**	....	
Subtract Immediate	subi	*wi,w2*	a3	Immediate	2,3	w2 ← (w2) – wi	.....	****	
Subtract Quick	subq	*wq,w2*	a2	Quick	1	w2 ← (w2) – wq	.....	****	
Subtract Single Floating	subs	s1,s2	21	Register	1	s2 ← (s2) – (s1)	** **	....	
Subtract Word	subw	*w1,w2*	a0	Register	1	w2 ← (w2) – (w1)	.....	****	
Subtract Word with Carry	subwc	*w1,w2*	91	Register	1	w2 ← (w2) – (w1) – C	.....	****	
Multiply Double Floating	muld	*d1,d2*	2a	Register	1	d2 ← (d2) × (d1)	*****	....	
Multiply Single Floating	muls	*s1,s2*	28	Register	1	s2 ← (s2) × (s1)	*****	....	
Multiply Word	mulw	*w1,w2*	98	Register	1	w2 ← (w2) × (w1)	.....	0*00	
Multiply Word Unsigned	mulwu	*w1,w2*	9a	Register	1	w2 ← (w2) × (w1)	.....	0*00	
Multiply Word Unsigned Extended	mulwux	*w1,l2*	9b	Register	1	l2 ← (w2) × (w1)	.....	0*00	
Multiply Word Extended	mulwx	*w1,l2*	99	Register	1	l2 ← (w2) × (w1)	.....	0*00	
Divide Double Floating	divd	*d1,d2*	2b	Register	1	d2 ← (d2) ÷ (d1)	*****	....	
Divide Single Floating	divs	*s1,s2*	29	Register	1	s2 ← (s2) ÷ (s1)	*****	....	
Divide Word	divw	*w1,w2*	9c	Register	1	w2 ← (w2) ÷ (w1)	.....	**00	D
Divide Word Unsigned	divwu	*w1,w2*	9e	Register	1	w2 ← (w2) ÷ (w1)	.....	0000	D
Negate Double Floating	negd	*d1,d2*	b4 3b	Macro	2	d2 ← (d1)	.....	....	
Negate Single Floating	negs	*s1,s2*	b4 3a	Macro	2	s2 ← (s1)	.....	....	
Negate Word	negw	*w1,w2*	93	Register	1	w2 ← (w1)	.....	****	
Modulus Word	modw	*w1,w2*	9d	Register	1	w2 ← (w2) MOD (w1)	.....	0*00	D
Modulus Word Unsigned	modwu	w1,w2	9f	Register	1	w2 ← (w2) MOD (w1)	.....	0000	D
Scale by, Double Floating	scalbd	*w1,d2*	b4 3d	Macro	2	d2 ← (d2) × 2(w1)	**.**	....	
Scale by, Single Floating	scalbs	*w1,s2*	b4 3c	Macro	2	s2 ← (s2) × 2(w1)	** **	....	

Legend: PSW Flags Field
- . = Flag not affected by instruction
- * = Flag set according to operation
- 0 = Flag set to 0
- 1 = Flag set to 1

Traps Field
- D = Divide-by-zero
- I = Illegal instruction
- M = Memory fault
- P = Page fault
- R = Read protect fault
- S = Supervisor only (privileged) instruction
- W = Write protect fault

TABLE B.1 Instruction Set By Function (Continued)

LOGICAL INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
And Immediate	andi	*wi,w2*	8b	Immediate	2,3	w2 ← (w2) & wi	..... 00**	
And Word	andw	*w1,w2*	88	Register	1	w2 ← (w2) & (w1)	..... 00**	
Or Immediate	ori	*wi,w2*	8f	Immediate	2,3	w2 ← (w2) | wi	..... 00**	
Or Word	orw	*w1,w2*	8c	Register	1	w2 ← (w2) |(w1)	..... 00**	
Exclusive-OR Immediate	xori	*wi,w2*	ab	Immediate	2,3	w2 ← (w2) ⊕ wi	..... 00**	
Exclusive-OR Word	xorw	*w1,w2*	a8	Register	1	w2 ← (w2) ⊕ (w1)	..... 00**	
Not Word	notw	*w1,w2*	ac	Register	1	w2 ← ~ (w1)	..... 00**	
Not Quick	notq	*wq,w2*	ae	Quick	1	w2 ← ~ wq	 0001	

SHIFT/ROTATE INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Shift Arithmetic Immediate	shai	*wi,w2*	38	Immediate	2	w2 ← (w2) SHA wi	..... 0***	
Shift Arithmetic Longword	shal	*w1,l2*	31	Register	1	l2 ← (l2) SHA (w1)	..... 0***	
Shift Arithmetic Longword Immediate	shali	*wi,l2*	39	Immediate	2	l2 ← (l2) SHA wi	..... 0***	
Shift Arithmetic Word	shaw	*w1,w2*	30	Register	1	w2 ← (w2) SHA (w1)	..... 0***	
Shift Logical Immediate	shli	*wi,w2*	3a	Immediate	2	w2 ← (w2) SHL wi	..... 00**	
Shift Logical Longword	shll	*w1,l2*	33	Register	1	l2 ← (l2) SHL (w1)	..... 00**	
Shift Logical Longword Immediate	shlli	*wi,l2*	3b	Immediate	2	l2 ← (l2) SHL wi	..... 00**	
Shift Logical Word	shlw	*w1,w2*	32	Register	1	w2 ← (w2) SHL (w1)	..... 00**	
Rotate Immediate	roti	*wi,w2*	3c	Immediate	2	w2 ← (w2) ROT wi	..... 00**	
Rotate Longword	rotl	*w1,l2*	35	Register	1	l2 ← (l2) ROT (w1)	..... 00**	
Rotate Longword Immediate	rotli	*wi,l2*	3d	Immediate	2	l2 ← (l2) ROT wi	..... 00**	
Rotate Word	rotw	*w1,w2*	34	Register	1	w2 ← (w2) ROT (w1)	..... 00**	

CONVERSION INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Convert Double Floating to Single	cnvds	*d1,s2*	b4 39	Macro	2	s2 ← (d1)	**.**	
Convert Double Floating to Word	cnvdw	*d1,w2*	b4 34	Macro	2	w2 ← (d1)	*....	
Convert Rounding Double to Word	cnvrdw	*d1,w2*	b4 35	Macro	2	w2 ← (d1)	*....	
Convert Rounding Single to Word	cnvrsw	*s1,w2*	b4 31	Macro	2	w2 ← (s1)	*....	
Convert Single Floating to Double	cnvsd	*s1,d2*	b4 38	Macro	2	d2 ← (s1)	*....	
Convert Single Floating to Word	cnvsw	*s1,w2*	b4 30	Macro	2	w2 ← (s1)	*....	
Convert Truncating Double to word	cnvtdw	*d1,w2*	b4 36	Macro	2	w2 ← (d1)	*....	
Convert Truncating Single to word	cnvtsw	*s1,w2*	b4 32	Macro	2	w2 ← (s1)	*....	
Convert Word to Double Floating	cnvwd	*w1,d2*	b4 37	Macro	2	d2 ← (w1)	....	
Convert Word to Single Floating	cnvws	*w1,s2*	b4 33	Macro	2	s2 ← (w1)	.... *	

Legend: PSW Flags Field
- . = Flag not affected by instruction
- * = Flag set according to operation
- 0 = Flag set to 0
- 1 = Flag set to 1

Traps Field
- D = Divide-by-zero
- I = Illegal instruction
- M = Memory fault
- P = Page fault
- R = Read protect fault
- S = Supervisor only (privileged) instruction
- W = Write protect fault

B

TABLE B.1 Instruction Set By Function (Continued)

COMPARE AND TEST INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Compare Double Floating	cmpd	*d1,d2*	27	Register	1	(d2) – (d1)	..... 00**	
Compare Immediate	cmpi	*wi,w2*	a7	Immediate	2,3	(w2) – wi	..... ****	
Compare Quick	cmpq	*wq,w2*	a6	Quick	1	(w2) – wq	..... ****	
Compare Single Floating	cmps	*s1,s2*	25	Register	1	(s2) – (s1)	..... 00**	
Compare Word	cmpw	*w1,w2*	a4	Register	1	(w2) – (w1)	..... ****	
Test and Set	tsts	*wa,w2*	72,73	Address	1	w2 ← (wa), wa<31> ← 1	....	P,R,W,M

CHARACTER STRING INSTRUCTIONS

Instruction Name	Syntax	Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Compare Characters r0 = length, r1 = string1, r2 = string2	cmpc	b4 0f	Macro	2	WHILE [(r0) = 0] & [((r2)) = ((r1))], r0 ← (r0) – 1, r1 ← (r1) + 1 r2 ← (r2) + 1	..... ****	P,R,M
Initialize Characters r0 = length, r1 = string1, r2 = string2	initc	b4 0e	Macro	2	WHILE (r1) ← (r2<7:0>) r0 ← (r0) – 1, r1 ← (r1) + 1 r2 ← (r2) ROT – 8	....	P,W,M
Move Characters r0 = length, r1 = string1, r2 = string2	movc	b4 0d	Macro	2	WHILE (r0) = 0, (r2) ← ((r1)) r0 ← (r0) – 1, r1 ← (r1) + 1 r2 ← (r2) + 1	....	P,R,W,M,

STACK MANIPULATION INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Pop Word	popw	*w1,w2*	16	Register	1	w1 ← (w1) + 4 w2 ← ((w2) – 4)	....	P,R,M
Push Word	pushw	*w2,w1*	14	Register	1	w1 ← (w1) – 4 (w1) ← (w2)	....	P,W,M
Restore Registers *fn*: f7	restd*n*		b4 28 ... b4 2F	Macro	2	*fn* : f7 ← ((r15)) : ((r15) + 8 × [7-*n*]) r15 ← (r15) + 8 × [8-*n*]	....	P,R,M,
Restore User Registers (privileged)	restur	*w1*	b6 03	Macro	2	r0 : r15 ← ((w1)) : ((w1) + 60)	....	P,R,M,S
Restore Registers *rn*: r14	restw*n*		b4 10 ... b4 1C	Macro	2	*rn* : r14 ← ((r15)) : ((r15) + 4 × [14-*n*]), r15 ← (r15) + 4 × [15-*n*]	....	
Save Registers *fn*: f7	saved*n*		b4 20 ... b4 27	Macro	2	(r15) – 8 × [8-*n*] : (r15) – 8 ← (*fn*) : (f7) r15 ← (r15) – 8 × [8-*n*]	....	P,W,M
Save User Registers (privileged)	saveur	*w1*	b6 02	Macro	2	(w1) – 4 : (w1) – 64 ← (r14) (r0)	....	P,W,M,S
Save Registers *rn*: r14	savew*n*		b4 00 ... b4 0C	Macro	2	(r15) – 4 × [15-*n*] : (r15) – 4 ← (*rn*) : ← (r14), r15 ← (r15)- 4× [4 *n*]	....	P,W,M

Legend: PSW Flags Field
- . = Flag not affected by instruction
- * = Flag set according to operation
- 0 = Flag set to 0
- 1 = Flag set to 1

Traps Field
- D = Divide-by-zero
- I = Illegal instruction
- M = Memory fault
- P = Page fault
- R = Read protect fault
- S = Supervisor only (privileged) instruction
- W = Write protect fault

TABLE B.1 Instruction Set By Function (Continued)

CONTROL INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX CVZN	Traps
Branch Conditional	b*	*ha*	48,49	Address	1-4	IF cond, PC ← ha	....	
Branch Floating Exception	bf*	*ha*	4c,4d	Address	1-4	IF cond, PC ← ha	....	
Call Subroutine	call	*w2,ha*	44,45	Address	1-4	w2 ← (w2) – 4, (w2) ← (PC), PC ← ha	....	P,W,M,
Call Supervisor	calls	*bb*	12	Control	1	trap 400 + 8 x bb<7:0>	....	
No Operation	noop	*bb*	00	Control	1	none	....	
Return From Subroutine	ret	*w2*	13	Register	1	PC ← ((w2)), w2 ← (w2) + 4	....	P,R,M
Return From Interrupt (privileged)	reti	*w1*	b6 04	Macro	2	Restore SSW, PSW and PC	....	S
Trap on Floating Unordered	trap*n*		b4 3e	Macro	2	IF PSW<ZN> indicated unordered, illegal instruction trap	....	I
Wait for Interrupt (privileged)	wait		b6 05	Macro	2	Wait for interrupt	....	S

Legend: PSW Flags Field
. = Flag not affected by instruction
* = Flag set according to operation
0 = Flag set to 0
1 = Flag set to 1

Traps Field
D = Divide-by-zero
I = Illegal instruction
M = Memory fault
P = Page fault
R = Read protect fault
S = Supervisor only (privileged) instruction
W = Write protect fault

TABLE B.2 Instruction Opcode/Mnemonic Summary

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	noop															
1	movwp	movpw	calls	ret	pushw		popw									
2	adds	subs	addd	subd	movs	cmps	movd	cmpd	muls	divs	muld	divd	movsw	movws	movdl	movld
3	shaw	shal	shlw	shll	rotw	rotl			shai	shali	shli	shlli	roti	rotli		
4					call				b* see Table 10.7				bf* see Table 10.8			
5																
6	loadw		loada		loads		loadd		loadb		loadbu		loadh		loadhu	
7	storw		tsts		stors		stord		storb				storh			
8	addw		addq	addi	movw		loadq	loadi	andw			andi	orw			ori
9	addwc	subwc		negw					mulw	mulwx	mulwu	mulwux	divw	modw	divwu	modwu
A	subw		subq	subi	cmpw		cmpq	cmpi	xorw			xori	notw		notq	
B					macros see Table B.3		privileged macros see Table B.4									
C																
D																
E																
F																

TABLE B.3 Macro Instruction Code Field (Opcode B4)

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	savew0	savew1	savew2	savew3	savew4	savew5	savew6	savew7	savew8	savew9	savew10	savew11	savew12	movc	initc	cmpc
1	restw0	restw1	restw2	restw3	restw4	restw5	restw6	restw7	restw8	restw9	restw10	restw11	restw12			
2	saved0	saved1	saved2	saved3	saved4	saved5	saved6	saved7	restd0	restd1	restd2	restd3	restd4	restd5	restd6	restd7
3	cnvsw	cnvrsw	cnvtsw	cnvws	cnvdw	cnvrdw	cnvtdw	cnvwd	cnvsd	cnvds	negs	negds	scalbs	scalbd	trapfn	loadfs

TABLE B.4 Privileged Macro Instruction Code Field (Opcode B6)

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	movus	movsu	saveur	restur	reti	wait										

Appendix C
Questions and Answers

1. Question
In virtual memory management, how is the S field in the page table entry used?

Answer
The two-bit S field is reserved for software, and may be used at the systems programmer's discretion. As an example, for valid pages it could be used to characterize a real page, or a virtual page that's mapped on top of an I/O space page. In the case of invalid pages, it could indicate that a page is on disk, in which case the RA field would point to the page's location in swap space. It might indicate that a zeroed page of memory should be allocated if the page is accessed. It might also indicate that the page is not in the virtual address space, so that any reference is illegal.

2. Question
Can software modify the U (Used) flag in either the TLB or cache?

Answer
The U flag in the TLB can be directly written by software; the U flag in the cache cannot be directly written, but can be reset by writing to the CAMMU Reset register.

3. Question
How do I read and write CAMMU registers?

Answer
CAMMU registers are read and written using memory loads and stores. The CLIPPER CPU accesses the D-CAMMU at virtual address 4800-49FF hex, and the I-CAMMU at virtual address 4A00-4BFF hex.

In systems utilizing two or more CLIPPER Modules, a CPU can execute global writes to all CAMMUs other than its companion D-CAMMU by accessing virtual address locations 4Cnn hex (global TLB writes), and 4Dnn hex (global register writes and TLB/cache resets). I/O can execute global writes to all system CAMMUs by accessing virtual addresses Cnn and Dnn hex.

CAMMU addressing and data formats are explained in detail in Chapter 3.

4. Question
Why are there two different branch on condition instructions, **bc*** and **br***?

Answer
While there are two sets of branch on condition mnemonics for relational predicates, there is only one set of branch on condition instructions in the hardware. The **bc*** mnemonics are used when the condition codes to be tested were set by a compare instruction, for example, **cmpq** or **cmps**. The **br*** mnemonics are used when the condition codes to be tested were set by a move or logical instruction (those that set only the N or Z condition codes). The **br*** mnemonics are designed to support an implicit comparison of a computational result with 0.

In general, two numbers are compared by subtracting them, then interpreting the condition codes appropriately. After a move or logical instruction with the result A, the condition codes are set as if the subtraction A-0 had been done, and the **br*** mnemonics interpret the codes as a comparison of A with 0. But after a compare instruction comparing A with B, the codes are set based on the subtraction B-A, the opposite sense expected by the **br*** mnemonics. The **bc*** mnemonics are simply a renaming of the **br*** mnemonics to account for the swap.

Some other architectures are able to use a single set of mnemonics that correspond to the CLIP-

PER **br*** mnemonics, but at the expense of extra hardware in a time-critical area that essentially swaps the operands to the compare instruction before performing the internal subtraction.

5. Question
With CLIPPER's very fast cycle time, how can nine addessing modes be implemented without using a special address calculation ALU?

Answer
CLIPPER has a load/store architecture, in which none of the instructions with addressing modes require the ALU to do anything except compute an effective address. The ALU therefore is used as the address calculation ALU with only a slight degradation of performance, because instructions with memory operands do not require the ALU for operations other than address calculation.

6. Question
How does the CPU instruction buffer work? Is a parcel fetched each time one is used, or is the buffer completely emptied before it is refilled?

Answer
The instruction buffer, shown in Figure C.1, contains a maximum of four 16-bit instruction parcels. Instructions are pipelined through the instruction buffer such that during each MCLK (30 ns) period the instruction parcel in PB is transferred to the B stage of the instruction pipeline, the parcel in PA is transferred to PB, the parcel in PL is transferred to PA, and the parcel in PH is transferred to PL if it has not yet been transferred. If the parcel in PH was transferred to PL the previous clock, the next instruction word (two parcels) is transferred from the I-CAMMU into PH and PL. A new instruction is therefore fetched from the I-CAMMU to the Instruction buffer every two MCLK cycles.

Table C.1 illustrates instruction pipelining through the instruction buffer. The table shows an instruction word consisting of parcels P2:P1 fetched from the I-CAMMU at clock n, and the transfer of each parcel during successive MCLKs. Note that at clock n+2, the last of the initial instruction parcels (P2:P1) has been transferred to the IB, and a new instruction word (P4:P3) is fetched from the I-CAMMU.

Table C.1 Instruction Buffer Pipelining

	clock n	clock n+1	clock n+2	clock n+3
PH:PL	P2:P1	X:P2	P4:P3	X:P4
PB	X	P1	P2	P3
PA	X	X	P1	P2

X = don't care

This is an elementary explanation of the instruction buffer. It is important to note that a direct path exists from each of the four registers to the B stage to eliminate the delay that would otherwise be caused by an empty instruction buffer, resulting, for instance, from a branch instruction.

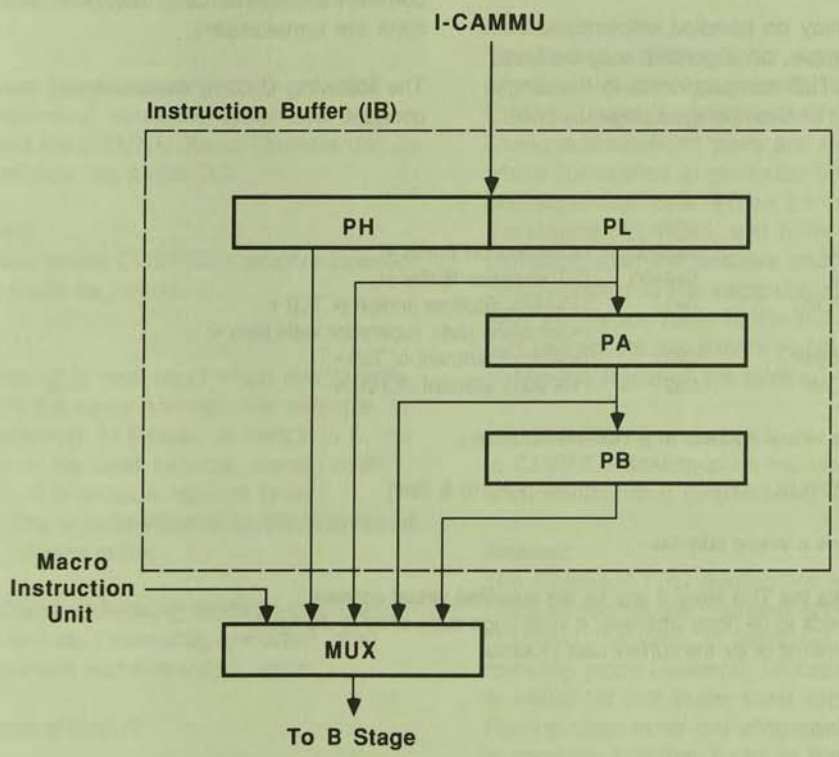
7. Question
How does the quadword buffer operate? Why does an access to it require less time than an access to the cache?

Answer
The quadword buffer holds the quadword from the most recently read cache line. The virtual address associated with the data in the quadword buffer is saved by CAMMU logic. When the CAMMU receives a read request, the virtual address is compared with the address associated with the data in the quadword buffer. This comparison takes place concurrently with the access to the cache and TLB. If there is a match, the data is read directly from the quadword buffer. This operation is inherently faster than completing a search of the cache, transferring data from the cache to the quadword buffer, then transferring data from the quadword buffer to the CPU.

8. Question
Is it possible for both caches to contain the same data? What happens if an I/O device executes a write to data which is valid in both CAMMUs? In this case, does the line containing the data get invalidated properly in each CAMMU?

Answer
It is possible for both caches to contain the same

Figure C.1 **Instruction Buffer**



data. This can occur when the operating system allows a process to map instruction and data pages to the same real pages (for self-modifying code), and data is written to the shared pages. If an I/O device or D-CAMMU writes to a location in a shared page that is cached in both CAMMUs, each CAMMU with Watch CPU Writes or Watch I/O Writes enabled updates its cache if the write is a single word, halfword, or byte, and invalidates the hit line if the write is a quad-word. In this way, consistency between the two caches is maintained.

9. Question
If a memory error is detected during a DAT (dynamic address translation), is the address saved?

Answer
The address is not saved by the CLIPPER Module. The memory interface or other system hardware must save the address in a snapshot register, if the real address causing the memory error is required.

10. Question
Does the TLB have enough information about memory accesses to invalidate its entries when the CPU changes corresponding entries in the page tables? Or must the CPU search the TLB each time it changes a page in order to manually invalidate an affected line?

Answer
The TLB does not have enough information about

memory accesses to invalidate page table entries changed by the CPU. But no search of the TLB is required to invalidate a changed page.

This invalidation may be handled efficiently in software. For example, an algorithm may be used to invalidate both TLB compartments in the single line corresponding to the changed page, in both the instruction and data caches. The method used guarantees that the old entries are invalidated, while reducing the potential for invalidating current translations. Long searches and comparisons are unnecessary.

The following C code demonstrates the basic concept:

```
#define TLBD            0x4800    /*Translation Buffer */
#define TLBI            0x4a00    /*Translation Buffer */
#define VAPORT          0x1       /*Virtual address portion of TLB */
                                  /*(contains user, supervisor valid bits) */
#define WCOMP           0x0       /*W compartment of TLB */
#define XCOMP           0x2       /*X compartment of TLB */

 /*Convert a virtual address to a TLB line address */

#define VATOTLBL(va)              (((va)  >> 10) & 0xfc)

 /*  Invalidate a virtual address.
  *
  *  Invalidate the TLB entry, if any, for the specified virtual address.
  *  This needs to be done whenever a valid page table entry is changed
  *  for the kernel or for the current user process.
  */
inval(vaddr)
register int vaddr;
{
 /*  Zero out the TLB lines, two for I-cache, two for D-cache.
  *
  *  The addition produces an address which is not integer aligned,
  *  but we still want to do a full-word store. For the memory-mapped
  *  TLB, this is correct.
  */
        vaddr = VATOTLBL(vaddr);
        *(int *)  (vaddr + TLBI  + VAPORT + WCOMP) = 0;
        *(int *)  (vaddr + TLBI  + VAPORT + XCOMP) = 0;
        *(int *)  (vaddr + TLBD  + VAPORT + WCOMP) = 0;
        *(int *)  (vaddr + TLBD  + VAPORT + XCOMP) = 0;
}
```

Not all modifications to a page table entry require this invalidation. For example, changing an entry from invalid (causing a page fault) to valid is OK, because the TLB only remembers valid transactions.

Even this step is unnecessary when a context switch is performed, since the whole user map is changed, and the CAMMU Reset Register can be used to invalidate the entire TLB.

11. Question
In what cases would CLIPPER's relative indexed addressing mode be used?

Answer
Relative indexing is best used when two variable items go into the same address. For example, in the case statement in Pascal, or switch in C, the first variable is the case selector, usually multiplied by 2 or 4 to index a table of branch addresses. The second variable is the address of the branch address table.

Relative indexed addressing mode is also useful to compiler writers. Optimizing compilers often use a 3-argument add instruction, as in

```
loada [r3](r6),r2
```

which signifies (r3)+(r6) → r2.

This addressing mode is also used for referencing two-dimensional arrays, when both indices are variable.

12. Question
In CLIPPER floating-point operations, how is rebiasing of the exponent on a trapped overflow or underflow handled?

Answer
The CLIPPER FPU does not rebias the exponents, since a trap handler is being summoned anyway. Handling this rebiasing is left for the relevant trap routine, as specified in the exception handling section of the *IEEE 754 Standard for Binary Floating-Point Arithmetic*, 1984.

13. Question
What is the role of the Exception Vector Table in exception handling?

Answer
The Exception Vector Table, located in the first real page of memory, contains the address and SSW value for each exception handler routine. These address/SSW pairs are stored in locations which correspond to particular traps, interrupts, and supervisor calls. When an exception occurs, the current PC, PSW, and SSW are saved on the stack; then the address and SSW value corresponding to the exception are transferred from the Vector Table to the PC and SSW registers (except for the SSW's P bit). Figure 2.9 shows the layout of the table.

14. Question
In CLIPPER floating-point calculations, how is accurate rounding guaranteed?

Answer
The CLIPPER FPU guarantees for +, −, ×, and / that all results are rounded according to IEEE Standard 754-1985. In particular, for the default rounding mode (nearest), all results are accurate to within 1/2 unit in the least significant place. For the other three rounding modes, the answer is accurate to within 1 unit in the least significant place.

15. Question
What type of devices can be used for interfacing to CLIPPER I/O signals?

Answer
The CLIPPER Module, main memory, I/O, and all other CLIPPER subsystems interface to the CLIPPER Bus. "Fast" devices, such as the 74F245 transceivers, are typically used for interfacing to the CLIPPER Bus. However, other technology devices can be used as long as they meet CLIPPER Module specifications for the particular CLIPPER system BCLK frequency.

16. Question
Is the CLIPPER Bus a multiplexed bus?

Answer
Yes. The Address/Data bus within the CLIPPER Bus is a 32-bit, bi-directional, multiplexed bus.

17. Question
What is meant by the term "synchronous" as applied to the CLIPPER Bus?

Answer
The CLIPPER Bus is a synchronous bus. This means that all input signals to the CLIPPER Module are sampled on BCLK positive edges and must meet the required setup times. Signals are not allowed to transition within the Tsu (setup time) of BCLK positive edges or undefined results can occur.

18. Question
Do CLIPPER Bus error signals generate traps?

Answer
There are three defined error signals associated with the CLIPPER Bus. They are $\overline{\text{MSBE}}$ (Memory Single Bit Error), $\overline{\text{MMBE}}$ (Memory Multiple Bit Error), and $\overline{\text{BERR}}$ (Bus Error). $\overline{\text{MSBE}}$ and $\overline{\text{MMBE}}$ both generate traps. $\overline{\text{BERR}}$ does not generate a trap. The $\overline{\text{BERR}}$ signal should be tied to the external interrupt control logic to produce an interrupt. Chapters 7 & 8 describe these signals in detail.

In one typical design, a device on the CLIPPER Bus is used to watch cycles for a timeout. In the case of a timeout, typically for a hung bus, the $\overline{\text{BERR}}$ signal is input to the interrupt controller and the CAMMU. The CAMMU aborts the current cycle, and the interrupt controller signals an interrupt.

19. Question
What happens to cache data if $\overline{\text{BERR}}$ is asserted during a data transfer?

Answer
The cache is affected by assertion of $\overline{\text{BERR}}$ during a data transfer to and from cacheable pages according to cache hit or miss and data caching policy as follows:

a. Load, cache miss, write-through and copy-back data:

   The quadword containing the data is written into a cache line, and the line is invalidated.

b. Load, cache hit, write-through and copy-back data:

   Data is loaded from the cache, so no bus access is required.

c. Store, cache miss, copy-back data:

   The quadword containing the data which has been stored to main memory is written into a cache line, and the line is invalidated.

d. Store, cache miss, write-through data:

   The cache is not affected.

e. Store, cache hit, write-through and copy-back data:

   The data is written into the cache.

20. Question
During quadword transfers, CT<1:0> are not used. Does the CAMMU set these two lines to a specific value for each quadword transfer?

Answer
Yes, the CAMMU always forces CT<1:0> = 00 during quadword bus operations.

21. Question
Does the CLIPPER Module release its bus request after $\overline{\text{MSBE}}$ or $\overline{\text{MMBE}}$ are asserted, prior to performing further bus operations?

Answer
Yes. CLIPPER completes the cycle normally (releases bus request) prior to performing another bus operation.

22. Question
If an I/O device is only accessing main memory space and is not using Slave I/O mode, must the I/O properly identify its operation on TG<2:0>? Or can these lines be set to arbitrary states?

Answer
The CAMMUs monitor TG<2:0> in order to perform bus watch selectively for just those operations which could affect system data inconsistencies as described in Chapter 7. This maximizes CLIPPER Bus efficiency by avoiding unnecessary bus watches. If the TG lines are hardwired to a specific value on the I/O interface, care should be taken to ensure that needed bus watch operations are not omitted. To ensure data consistency, the TG<2:0> code should be 010 (copy-back). This forces bus watch to be invoked during all I/O read and write operations. However this design can decrease CLIPPER performance, since it forces invocation of bus watch of I/O operations not requiring bus watch.

23. Question
What is the highest priority maskable interrupt?

Answer
Interrupt levels are prioritized in descending order, with level 0 having the highest priority. Interrupt numbers within levels are not prioritized. Therefore interrupt vector number 0X, indicating interrupt level 0 and interrupt number X (0–15) within level 0, has the highest priority.

24. Question
Is $\overline{IRQ}$, when asserted with interrupt level 0 number 0 on $\overline{IVEC}$, handled in the same way as an NMI (non-maskable interrupt)?

Answer
No. $\overline{IRQ}$ asserted with vector number 00 ($\overline{IVEC}$=00) is a maskable interrupt; that is, it can be masked by clearing the EI flag in the SSW. $\overline{NMI}$ cannot be masked. However, if EI = 1, then vector number 00 will vector to location 800 hex in the Exception Vector Table, which contains the NMI exception vector entry.

25. Question
How long does it take to execute the CLIPPER Module's internal diagnostics? What is the module's behavior when no error conditions are detected by the diagnostics?

Answer
The current maximum duration of internal diagnostics is 4500 MCLKs. However, 10,000 MCLKs should be allowed to accommodate future additions to the diagnostic routines.

If error conditions are detected during diagnostics, $\overline{URF}$ is asserted and CPU execution stops. If no errors are detected, the CPU begins execution at location 6000 hex of supervisor virtual address space.

26. Question
Can wait states be placed between each data word during a quadword transfer?

Answer
Yes, by toggling $\overline{RDY}$, for all quadword operations except Clear Dirty (see Chapters 7 & 8). Clear Dirty is an optional feature, selectable under software control. The Clear Dirty option allows the memory interface to be designed to update memory with data supplied by the CAMMU as a result of Watch I/O Reads operation. The CAMMU uses quadword transfers for these Clear Dirty operations, but does not monitor the $\overline{RDY}$ signal for wait states.

Wait states can be introduced during other quadword transfers by releasing $\overline{RDY}$ during data cycles of the quadword operations.

27. Question
Are CLIPPER's integer registers general-purpose?

Answer
All integer registers may be used as address, index, data or stack pointer registers. They may not, however, be used for floating-point data.

Floating-point registers can only be used for floating-point data. They cannot be used as address, index, or stack pointer registers.

28. Question
Can the 11 unused bits in the page table directory entries be used?

Answer
These bits are reserved by Fairchild. User software must program them to 0.

29. Question
How does CLIPPER support system calls which are required by an Operating System?

Answer
CLIPPER allows 128 possible system calls, each with its own address vector. The UNIX operating system, for example, requires only about 60 system calls.

CLIPPER has its own set of supervisor registers for such calls, so saving the user set of registers is not necessary during the context switch associated with a system call. The TLB entries have tag bits for supervisor valid entries which allow the TLB to be purged of just the user valid entries, leaving supervisor valid entries intact after a context switch.

30. Question
Can the cache be used in unmapped mode? What does unmapped mode do?

Answer
The CAMMU performs two system functions: caching and memory management.

The cache can be used in unmapped mode, with caching policy determined by the UST bits in the CAMMU Control Register.

The only memory management functions that are performed in unmapped mode are those performed by the HTLB. Except for addresses mapped through the HTLB, virtual addresses from the CPU are passed to the CLIPPER Bus as real addresses. The TLB is not used in unmapped mode. The HTLB does not provide protection checking in unmapped mode except in two cases:

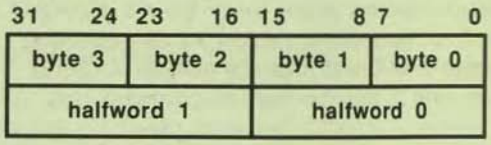
1. Instruction access to I/O space produces a protection error.
2. Test and Set to I/O space produces a protection error.

31. Question
Can differing caching policies be assigned to individual pages in unmapped mode?

Answer
No. Except in the case of HTLB entries, the TG signal lines, which identify caching policy, are controlled by the UST field in the CAMMU Control Register in unmapped mode. However, software may change the UST field at any time, which in effect assigns caching policies for individual pages.

32. Question
Does the CLIPPER Module dynamically adjust the size of the CLIPPER Bus for byte or halfword transfers?

Answer
No. Bytes and halfwords must be transferred to the CAMMU during read operations, and the CLIPPER Module always transfers data to the CLIPPER Bus during write operations as follows:



Byte/halfword alignment occurs on the CLIPPER Module as described below:

CPU load operation:
When the CPU executes a load instruction from main memory, the CAMMU reads the byte/halfword from main memory with the byte/halfword supplied on the data bus in the position shown above. If the data is placed in the cache, the byte/halfword will be placed in the corresponding byte/halfword of the appropriate cache word without affecting other bytes/halfword of that cache word. The byte/halfword is placed in the least significant byte/halfword of the destination CPU register. Figure C.2 shows a pictorial representation of this transfer for a load of byte 2.

Figure C.2 Load Byte Operation

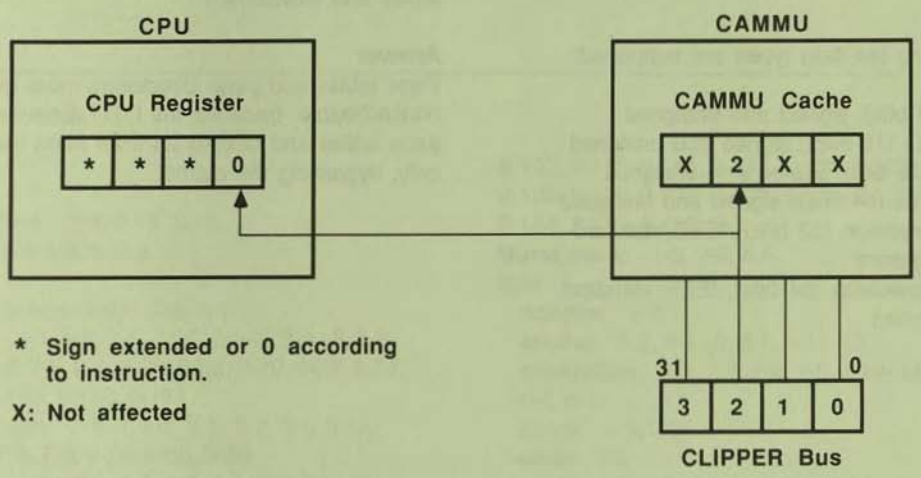
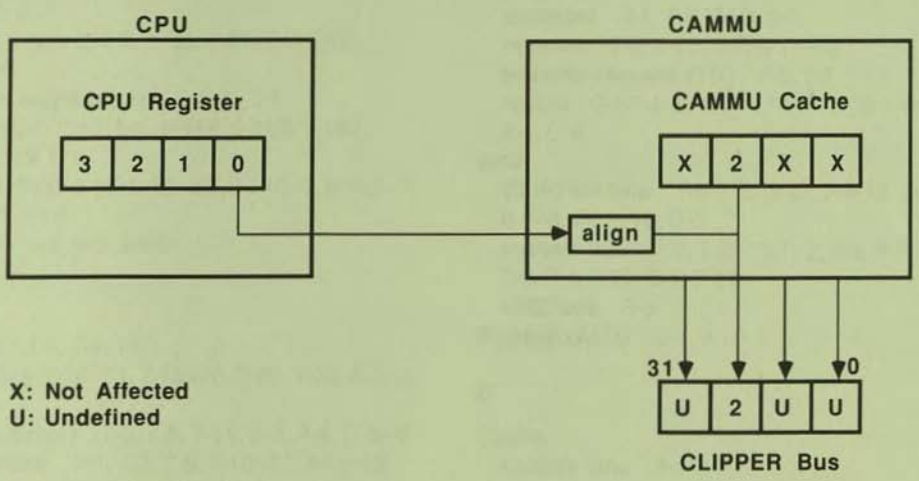


Figure C.3 Store Byte Operation



Store Operation:
If the data from the CPU is placed in the cache during a byte/halfword store to main memory, the appropriate byte/halfword of the cache is updated without affecting the other data locations of that word. The CAMMU presents the data on the appropriate CLIPPER Bus data lines as shown in the first diagram of this answer. The remaining data lines are undefined. Figure C.3 shows a pictorial representation of a byte 2 store to a cacheable, write-through page.

33. Question
If I read the PDO register, am I guaranteed zeros in the lowest 11 bits?

Answer
Yes, the lower eleven bits are zero.

34. Question
What data types are supported by CLIPPER?

Answer
The following ten data types are supported:

Bytes (8 bits); signed and unsigned
Halfwords (16 bits); signed and unsigned
Words (32 bits); signed and unsigned
Longwords (64 bits); signed and unsigned
Single-precision (32 bits); IEEE standard-floating-point
Double-precision (64 bits); IEEE standard floating-point

35. Question
What caching strategy should be used for page tables and directories?

Answer
Page tables and page directories must be noncacheable, because the DTU accesses the page tables and directories from main memory only, bypassing the cache.

Index

A

Access
  protection 2-6, 2-16, 3-12
  time 1-5, 1-9, 4-1–3
Address
  and data bus (AD) 7-4, A-1
  space 1-1, 1-8, 2-6, 2-16, 3-1–2, 3-4–5, 3-8, 3-10, 7-2, 7-5, 9-1, 9-28, 9-52, 9-70, 9-72, 9-74, 9-94, 9-138, 9-148, 9-154
  translation 1-5–7, 2-2, 2-6, 3-2, 3-5, 3-12, 3-14, 3-16, 7-5, 9-78, 9-80, 9-89
addressing mode 1-2, 1-8, 2-1, 2-7, 2-10–5, 2-21, 7-10, C-3
alignment 2-7, 6-18, 9-170–1, 9-179
ALU 2-2, 5-2–4, C-2
ANDing 6-2
ANSI 1-3, 1-6, 1-9, 6-6, 6-30, 6-31, 7-6, 7-10, 7-13–14, 9-6
Application architecture 1-1–2, 1-4
Array 1-8, 2-7, 2-15, 8-1, 9-156, 9-160, 9-165, 9-169, 9-171, 9-179
ASCII 6-9, 6-28, 9-16, 9-32–33, 9-110–1, 9-162–3, 9-166–7, 9-174–8
Assembler 1-9, 9-3, 9-26

B

Bandwidth 1-6, 7-1, 7-8
BCLK 7-2–4, 7-10–11, 7-13–16, 7-20, 7-22, 8-3–4, 8-8–12, A-4
$\overline{\text{BERR}}$ (Bus Error) 7-3, 7-8, 7-11, 8-3, A-4, C-5–6
BG (Bus Grant) 7-1, 7-3, 7-6, 7-10–11, 8-11–13, A-3
Boot
  general 3-1–2, 3-8, 3-10, 3-16, 7-5, 7-9, 7-10, 8-4, 9-1, 9-13, 9-16–29, 9-34–5, 9-52, 9-54–55, 9-57, 9-59, 9-70–73, 9-82–84, 9-96, 9-100, 9-156–7
  memory space 3-1–2, 7-9, 9-1
BR (Bus Request) 7-1, 7-3, 7-6, 7-10–11, 8-11–13
Branch 1-8, 2-1–2, 2-5, 2-8, 2-11, 2-21, 6-2–3, 6-22, 6-25, 7-10, 9-40, 9-76, 9-82, 9-84, 9-96, 9-102, 9-112, 9-114, 9-116, 9-120, 9-122, 9-124, 9-126, 9-128, 9-130, 9-132, 9-134, 9-136, 9-138, 9-144, 9-150, 9-152, C-1
Burst mode 1-6, 1-9, 4-6
Bus
  adapter 7-7
  arbiter 8-2, 8-4–5, 8-7, 8-11–13
  arbitration 1-9, 7-2, 7-6, 7-8, 7-10–13, 7-15, 8-7, 8-12
  clock 4-5, 7-3
  error 7-8
  grant 7-1, 7-3, 7-6, 7-10–11, 8-11–13
  I/O 7-7, 7-13, 8-2, 8-5–7
  master 4-7, 7-2, 7-4–15, 7-18–19, 8-2, 8-7, 8-11–12
  protocol 7-1, 7-6, 7-10, 8-5
  request 7-6, 7-10, 8-7, 8-11–13
  transfer request (TR) 7-3, 7-7, 7-12
  watch 3-17, 4-6–7, 7-4, 7-15–17, 8-1, 8-3, 9-1, C-6
also
  CLIPPER bus 1-6, 1-9, 3-17, 7-1–12, 7-20, 8-1–8, 8-12–13, C-5
  system bus 1-9, 2-20, 3-1–2, 3-8, 4-7, 7-1–2, 7-4, 7-6, 7-11, 8-4, 8-11
  VME bus 8-5
Bypass (ALU) 5-4, 7-10

C

Cache
  access time 4-1
  architecture 1-5
  compartment 3-12, 4-4, 4-8–9, 9-10
  design 1-5–6
  line replacement 7-7, 7-20–21
  lines 7-4–5
  memory 1-7, 4-1–2
  and MMU 1-5
  purge 7-5, 8-12, 9-13, 9-28, 9-30
  replacement time 4-5
Cacheable 3-16–17, 4-6, 7-5, 7-15, 8-7, 9-1, 9-11, 9-13, 9-16, 9-20, 9-27

Caching strategy
copy-back 3-16–17, 4-6–7, 4-9, 7-5–6, 7-13, 8-3, 9-10
noncacheable 3-16–17, 4-6, 7-5, 8-7, 9-1, 9-11, 9-13, 9-16, 9-27, C-10
write-through 3-16–17, 4-6, 7-5, 9-13, 9-28
CBSY (Cache Busy) 7-3, 7-7, 7-10, 7-14, 7-16, 7-18, 8-3–4, 8-11–13, A-2
Clear dirty 3-16, 8-2, 8-4, C-7
CLIPPER
bus (*see* **Bus**)
module 1-4–6, 1-9, 2-2, 2-5, 2-7, 2-16–7, 2-19, 2-22, 3-17–19, 7-1, 7-4–10, 8-1–2, 8-4, 8-6–9, 8-13, 8-15, 9-16, 9-27
CMOS 7-2
Compiler 1-1, 1-7–9, 2-1, 9-1, 9-156
Complex instruction set computer (CISC) 1-8, 2-1, 10-1
Condition code 2-2, 6-2–3, 6-6, 6-26, 6-33
Conditional
branch 6-2–3
trap 2-19
Consistency 1-6, 4-6–7
Context
general 1-8–9, 2-1, 2-6, 2-17, 6-3, 6-5, 9-10, 9-26, 9-42,9-46–47, 9-98, 9-102, 9-136, 9-138, 9-140, 9-142, 9-144, 9-156–7
switch 2-6, 2-17, 6-5, 9-10, 9-26, 9-42, 9-46–7, 9-102, 9-136, 9-156–7, C-4, C-7
Control register 2-2, 3-16–20, 4-7, 9-10, 9-27, C-8
Controller 1-9, 2-17, 4-7, 7-5, 7-7–8, 7-20, 8-2, 8-5–10
Copy-back (*see* **Caching strategy**)
CT (Cycle type) 7-3–4, 7-12–14, 7-18, 8-2–4, 8-13, A-2, C-6

D

Data type 2-1, 2-7, 2-15, 6-1, 6-9
Deadlock 7-8, 8-2, 8-7
Debug 1-1, 1-9
Demand paging 1-9
Denormal 6-8, 6-11–15, 6-22–23, 6-25, 6-29, 6-32–35
Diagnostic Trap Group 2-19, 2-22
Diagnostics 7-2–3, 7-9–10, C-7
DIR (Direction Control) 7-3, 7-6–7, 7-12–13, 8-13, A-1
Dirty
data 1-8, 2-6, 3-11, 3-13–14, 3-16–17, 4-9, 6-3, 7-9, 7-14, 7-20, 8-2–4, 9-7, 9-11, 9-13, 9-98
flag 2-6, 3-13–14, 6-3
(*see also* **Line dirty, Clear dirty**)
Displacement 2-11–15
Divide by zero 2-16, 2-20–21, 6-5–6, 6-9, 6-25–27, 6-32–33, 6-35, 9-127–8
DMA (direct memory access) 2-17, 4-7, 7-5
Double-precision 2-2, 2-7, 5-4, 6-1, 6-3–9, 6-14–17, 6-27–29, 6-32, 6-34–35, 7-10, 9-160, 9-170, 9-172, 9-178–9
DRAM 1-5, 4-1, 4-5, 8-3
DTU (Dynamic Translation Unit) 3-11, 3-12, 3-15, 3-16, 7-7, 7-19–20, 8-12, 9-80

E

Exception
floating-point, under/overflow (*see* **Floating-point**)
handling 2-1, 2-17, 9-32, 9-106, 9-148, 9-150–153
integer (*see* **Integer**)
invalid operation (*see* **Invalid operation exception**)
vector table 2-6, 2-17–18, 2-20, 2-22, 7-8, 8-4, 9-1–2, 9-17–19, 9-22, 9-26, 9-32, 9-38–39, 9-48, 9-60–61, 9-64, 9-112, 9-136, 9-144, C-7
Execution unit 1-2, 1-4, 1-6, 5-2, 5-3, 7-10
Exponent
biased 6-1, 6-7–8, 6-11–16, 6-28, 6-32, 6-34–5, C-5
wrap 6-34

F

Fault register 2-6, 2-20–21, 3-11, 3-16, 3-19–20, 7-8
Floating-point
inexact 2-17, 6-33, 6-35, 9-5, 9-63
invalid 2-17, 6-18, 6-24, 6-26–27, 6-33, 6-35, 9-5, 9-63
under/overflow 2-17, 6-33–5, 9-5, 9-63
exception 2-2, 2-19–21, 6-2–3, 6-33–34, 9-4
FORTRAN 1-2, 1-9, 6-30–31, 9-1
Fraction bit 2-2, 6-1, 6-4, 6-14

G

General-purpose registers 1-8, 2-1–2, 2-7–15, C-7
Global write 3-18, 7-4, 7-18–19, C-1

H

Halfword 2-2, 2-7–8, 7-4–5, 8-2–4, 9-3, 9-154–5
Hardware reset 2-2, 6-3, 9-18
Hardwired instructions 1-8, 2-7–8, 10-1
Hardwired TLB (HTLB) 2-6, 3-2, 3-8–13, 3-16–17, 7-5, 9-1, 9-16, 9-18–21, 9-24, 9-26, 9-28–29, 9-34, 9-55, 9-57–59, 9-84, 9-100, C-8
Hit rate 1-5, 1-9, 4-1–6, 5-3

I

$\overline{\text{IACK}}$ (Interrupt Acknowledge) 7-3, 7-7, 7-20, 7-22, 8-8–11, A-4
IEEE 754 standard 2-2, 5-4, 6-1, 6-6, 6-8, 6-34, C-5
Infinity arithmetic 6-9–10
I/O
  bus (*see* **Bus**)
  device 3-2, 3-16, 3-18, 7-2, 7-4–6, 7-15, 8-5, 9-2, 9-18
  space 3-1, 3-10, 3-13, 3-16–18, 7-5, 7-8, 7-18, 7-20, 8-7, 9-19, 9-26–29, 9-34–35, 9-52, 9-54–55, 9-57, 9-70–73, 9-82–83, 9-156–7
I/O processor (IOP) 8-2, 8-5–7
Instruction
  format 2-1, 2-8–10
  prefetching 1-6, 3-16–17, 4-4–5, 5-2–3, 7-1, 9-1, 9-10–11, 9-20, C-2
  set 1-1–5, 1-7–8, 2-1, 2-7, 10-1, B-1–6
  set architecture 2-1
Integer
  arithmetic 2-17, 2-19, 2-21, 5-3, 6-6, 9-60, 9-63, 9-112
  exceptions 6-5, 6-33
  overflow 6-5–6, 6-33
Integration (level of) 1-6–7, 6-1
Interbus arbitration 8-6
International Standards Organization (ISO) 2-16, 6-29–31
Interrupt
  general 1-9, 2-5–6, 2-16–19, 2-22–23, 3-16, 5-3, 6-3, 7-2–3, 7-7–8, 7-10–11, 7-20, 8-2, 8-6–10, 9-1–2, 9-7, 9-16–23, 9-27, 9-31–33, 9-38–39, 9-60–61, 9-64–67, 9-102–3, 9-112–3, 9-136–7, 9-144–7, C-7
  bus 1-9, 2-17, 2-23, 7-2, 7-7, 7-20
  controller 7-20, 8-2, 8-7–10
  level 2-5–6, 2-17, 2-22, 7-7–9, 9-7, 9-32, 9-60–61, 9-113, 9-137, 9-145
  non-maskable 2-17, 2-22–23, 7-3, 7-7, 9-102, 9-144–5, C-7
  number 2-5, 2-22, 7-7–8, 9-7
  request 2-22–23, 7-3, 7-7, 7-11, 7-20, 8-6, 8-8–10, C-7
  service 2-6, 2-19, 2-23, 7-11
  vector 1-9, 7-3, 7-8, 7-20, 8-7, 8-9, 9-31, 9-60, 9-65, C-7
INTRAP 2-5–6, 7-8, 2-16–17, 2-19, 2-22, 9-22, 9-32–33, 9-46–47, 9-58, 9-102, 9-106–7, 9-113–23, 9-125, 9-127, 9-129, 9-131–3, 9-135, 9-137, 9-139, 9-141, 9-143, 9-145, 9-148–9, 9-151–152, 9-180
Invalid operation exception 2-17, 2-20, 6-8, 6-18, 6-24, 6-26–29, 6-31–33, 6-35, 9-5, 9-128–9
$\overline{\text{IRQ}}$ (Interrupt Request) 2-22, 7-3, 7-7, 7-20, 7-22–23, 8-7–11, A-4, C-7
$\overline{\text{IVEC}}$ (Interrupt Vector) 2-17, 2-22, 7-3, 7-7, 7-20, 7-22, 8-7–10, 9-31, 9-39, 9-180, A-4, C-7

K

Knuth 9-160

L

Line dirty 8-4
Load/store 1-6, 1-8, 2-7–8
Locality 4-2
$\overline{\text{LOCK}}$ 7-3, 7-7, 7-20, 8-13, A-2
Longword 2-2, 2-7, 2-8, 6-5, 9-48

M

Macro instructions 1-1, 1-8, 2-7–8, 2-10, 2-17, 2-21–23, 5-2, 5-4, 6-1, 6-3, 7-3, 7-7–8, 7-20, 7-22, 7-23, 9-23, 9-66, 9-102–3, 10-1
Main memory space 3-1–2, 3-16–17, 7-5, 7-13, 8-4
Mapped mode 2-6, 3-2, 3-13, 7-5, 9-1, 9-7, 9-17, 9-20, 9-34, 9-38, 9-84
MCLK 7-4, 7-10
Memory
  errors 2-20–21, 7-8, 8-4, 9-114–5
  hierarchy 4-1
  interface 7-6–8, 7-14, 7-16, 8-2–4, 8-11, 8-13, 9-114
  mapping 2-5, 3-10, 9-1, 9-16–17, 9-34, 9-39, 9-87
  space system tag (*see also* **System tag** and **TG**) 7-2, 7-5
$\overline{\text{MMBE}}$ (Memory Multiple Bit Error) 2-20–21, 7-3, 7-8, 8-3–4, A-3, C-5
Motherboard 1-9

$\overline{\text{MSBE}}$/$\overline{\text{RETRY}}$ (Memory Single Bit Error/Retry) 2-20, 7-3, 7-8, 7-20, 8-4, 8-7, A-3, C-5–6
MULTIBUS 8-5
Multiplex 7-2, 7-4, 7-13, 8-3, 8-13

N

NaN (Not a Number) 6-3–4, 6-8–9, 6-11, 6-14–18, 6-22–33, 6-35, 9-32–33
$\overline{\text{NMI}}$ (Non-maskable Interrupt) 2-23, 7-3, 7-7–8, 7-20, 7-22–23, 9-23, 9-102–3, A-4
$\overline{\text{NMIACK}}$ (Non-maskable Interrupt Acknowledge) 7-3, 7-7, 7-20, 7-22, A-4
Non-maskable
  general 2-17, 2-22–23, 7-3, 7-7, 9-102, 9-144, 9-145
  interrupt (*see* **Interrupt**)
Noncacheable (*see* **Caching strategy**)
Normalization 6-18, 6-22, 6-34, (*see also* **Denormal**)
Null handler 9-16, 9-18, 9-20–23

O

Open drain 7-2, 7-4, 8-13
OSC (Oscillator Input) 7-2, 7-4, A-5
Overflow exception 2-20–21, 6-28, 6-34, (*see also* **Floating-point**)

P

Page
  directory 2-20–21, 3-16, 9-8, C-8, C-10
  directory origin (PDO) 2-6, 3-5, 3-8, 3-12, 3-16, 3-19–20, 9-8–9, 9-37, 9-46, 9-74, 9-78, 9-80, 9-180, C-9
  fault 2-16–21, 3-10–11, 5-3, 9-5, 9-13–17, 9-28, 9-50–51, 9-56, 9-58, 9-61, 9-63, 9-74, 9-79–80, 9-87, 9-89, 9-96–97, 9-116–117, 9-130–131, 9-149, 9-158, 9-161, 9-170
  table 1-9, 2-6, 2-20–21, 3-4–12, 3-16–17, 7-5, 7-7, 7-19–20, 9-1, 9-8, 9-10, 9-12–17 9-20, 9-28, 9-29, 9-31, 9-34–38 9-43, 9-48–59, 9-68, 9-69, 9-72–75, 9-78–80, 9-84, 9-88–97 9-119, 9-139, C-1, C-3–4, C-10
Parcel 2-8, 2-10–11, C-2
Pascal 1-2, 1-9, 6-29–31 9-1, C-5
Physical address 1-6, 9-78, 9-80
Priority 2-5–6, 2-17, 2-22–23, 7-6–8, 7-10, 7-20, 8-7–12
Private registers 1-8–9
Privileged 2-1, 2-8, 2-10, 2-16–17, 2-19, 2-21, 9-5, 9-63, 9-134, 9-135
Program
  counter (PC) 1-6, 1-8, 2-2, 2-11, 2-13–19, 2-22, 4-5, 6-3, 9-32–33, 9-46–47, 9-98, 9-106–7, 9-118–9, 9-124–9, 9-133, 9-156–7, 9-157, 9-180
  status word (PSW) 2-2, 2-16–22, 6-1–3, 6-6, 6-24, 6-26, 6-29–30, 6-33–35, 7-9, 9-1, 9-4–5, 9-16, 9-20–21, 9-33, 9-46–47, 9-52, 9-55, 9-70–71, 9-74–75, 9-78–79, 9-88–95, 9-102, 9-106–7, 9-118, 9-136, 9-138, 9-148–51, 9-165, 9-180–1
Protection
  code 2-6, 2-20–21
  fault 2-20–21, 3-16, 9-5, 9-86, 9-120–3, 9-130–1

Q

Quadword
  general 1-5, 4-1–2, 4-7–8, 7-4, 7-10, 7-12, 7-14–16, 7-20, 8-1–4, 8-11–12
  buffer 4-2, 4-7–8, C-2
  transfer (*see also* **Burst mode**) 7-4, 7-10, 8-2, 8-12

R

Random access memory (RAM) 1-5, 4-1, 4-5, 6-1, 8-1, 8-3–4, 9-16, 9-18, 9-20–21, 9-24–26, 9-160
RATE 4-2, 7-3–4, 7-16, A-4
RDY (Ready) 7-3, 7-6, 7-8, 7-10–16, 7-20, 8-3–4, 8- 11–13, A-3
Read-modify-write 7-7, 7-19, 8-12
Read only memory (ROM) 2-10, 2-17, 2-21–22, 3-1, 6-3, 8-4, 9-16, 9-23–24
Real address 2-6, 2-17, 3-1–2, 3-4, 3-7–8, 3-10, 3-12, 3-17, 3-19, 4-7–8, 7-5, 7-12, 7-14, 7-18, 8-4, 9-1–2, 9-10, 9-12–13, 9-16, 9-52–53, 9-59, 9-69, 9-73–74, 9-78–80
Reduced instruction set computer (RISC) 1-8, 2-1, 10-1
Referenced flag 3-14
Register-to-register 1-4, 1-8, 2-2, 6-1
Reserved value 6-3, 6-8
Reset 2-2, 2-4–6, 2-16, 3-1, 3-16–20, 6-3, 7-2–4, 7-9–10, 7-18, 9-1, 9-9–11, 9-16–18, 9-20–21, 9-38–39, 9-47, A-4
Reset register 2-6, 3-16–20
Resource manager 1-6, 5-2–3

RETRY (*see also* **MSBE/RETRY** and **Deadlock**) 7-3, 7-8, 7-19–21, 8-7
Rounding mode 2-2, 2-4, 6-2–3, 6-8–9, 6-19, 6-22–23, 6-26–35, 9-5, C-5
Runtime 9-42–43, 9-147

S

Self-test 7-8, 7-10
Set associative 3-12, 4-3
Setmap 9-39, 9-180
Signed integer 6-5–6, 6-11, 6-29–33
Simulator 1-1, 1-9
Sine 6-9
Single-word read/write 6-33, 7-12–16, 7-20–21, 8-4
Slave
  general 7-5–6, 7-8, 7-10, 7-12–15, 8-13, 9-13
  device 7-13
  I/O 7-5–6, C-6
Stack pointer (SP) 2-20, 7-8, 9-1, 9-16, 9-18–21, 9-47, 9-98–99, 9-102, 9-106, 9-114, 9-118, 9-136–7, 9-156–7, 9-170–1, 9-173, 9-180–1
Stale data (*see also* **Bus watch**) 4-7
Sticky flag 6-2, 6-33, 6-34
String 1-8, 2-1–2, 2-8, 2-22, 6-1, 6-9, 6-28, 9-22–23, 9-27, 9-38, 9-44–45, 9-67, 9-104, 9-108, 9-110, 9-142–3, 9-163, 9-166–7, 9-174–8
Supercomputer 1-1–8, 2-1, 5-1
Supervisor
  mode 1-8, 2-2, 2-5–6, 2-5, 2-16–17, 3-2, 3-5, 3-8, 3-13, 3-16, 6-1, 7-10, 9-1–2, 9-17, 9-20, 9-22, 9-106, 9-116–7, 9-120, 9-122, 9-124, 9-126, 9-128, 9-130, 9-132, 9-136–7, 9-156
  registers 2-1–2, C-7
  stack 2-17, 2-19–21, 9-1–2, 9-16, 9-20–21, 9-24, 9-27, 9-39, 9-46, 9-57–58, 9-102, 9-106
Synchronous 1-9, 7-1, 7-4
System
  bus (*see* **Bus**)
  call 1-8–9, 2-1, 6-34, 9-1–2, 9-16–18, 9-22, 9-26–27, 9-31–33, 9-38–39, 9-42, 9-60, 9-64–65, 9-86–87, 9-90, 9-102, 9-118, 9-120, 9-122, 9-136–43, 9-158, 9-163–4, C-7
  status word (SSW) 2-2, 2-5–6, 2-16–22, 3-2, 3-4, 3-8, 3-17, 6-1, 6-3, 7-7, 7-9, 9-1, 9-6–7, 9-16, 9-18, 9-20–23, 9-32–33, 9-38–39, 9-46–47, 9-60–61, 9-63–65, 9-98, 9-102–3, 9-106–7, 9-112–3, 9-116–7, 9-120–3, 9-132–7, 9-144–5, 9-150–1, 9-154–5, 9-180
  tag 3-1–2, 3-8, 3-13, 3-16–17, 3-20, 7-2–3, 7-5, 7-12, 8-2, 8-7, 9-10–11, 9-13, 9-18, 9-20, 9-28, 9-52–53, 9-68–69, 9-72–73, 9-78–79, 9-82–83, 9-89, 9-91, 9-93, 9-96–97

T

Test and set 2-8, 7-17, 7-19
TG (Memory space system tag) 3-16–17, 7-3, 7-5–6, 7-8, 7-12–14, 8-2–4, 8-7, 8-13, A-1
Timing 2-23, 4-6, 7-2, 7-5–12, 7-14, 7-16, 7-19–23, 8-1, 8-3, 8-9–10, 8-12
TR (Transfer Request) 7-3, 7-7, 7-12, 7-14, 8-2–3, 8-13, A-3
Trace trap 2-2, 2-4, 2-6, 2-17, 2-22, 7-9, 9-5, 9-7, 9-63, 9-134–5
Trap handler 2-2, 2-19–22, 6-3, 6-28, 6-34–35, 9-16, 9-22–23, 9-86–87, 9-112–6, 9-118, 9-120–6, 9-128, 9-130, 9-132, 9-134, 9-148, 9-156, 9-158
Trap status 2-2, 2-4–5, 2-20–22, 9-4–5, 9-148–50
Truncation 6-2, 6-5, 6-17, 6-28, 6-30–31

U

ULTRIX 1-9
Uncorrectable memory error 2-20, 9-114
UNIX 1-1–4, 9-1, C-7
Unmapped mode 2-6, 3-2, 3-13, 7-5, 9-1, 9-84, C-8
URDIAG (Apply Diagnostics) 7-3, 7-10, A-5, C-7
URF (Unrecoverable Fault) 2-19, 7-3, 7-8–10, A-5, C-7
User mode 1-8, 2-6, 2-16–17, 2-21, 3-2, 3-8, 3-16, 7-10, 9-7, 9-17, 9-41–42, 9-46–47, 9-116–7, 9-121, 9-123, 9-132–3, 9-151, 9-163
User registers 2-1, 3-2, 9-98–99
UST 3-17, 7-5, 9-10–11, 9-20–21, 9-27, 9-118, C-8

V

VAX 1-1, 1-9
Vector
  number 2-22, 7-7–8, 7-20, 7-22, 8-2, 8-7–10
  table (*see* **Exception vector table**)
Virtual
  address 1-8, 2-6, 2-20, 3-1–19, 4-7–8, 9-10, 9-14–16, 9-20, 9-28, 9-30–31, 9-34, 9-46, 9-58–59, 9-74–75, 9-78–80, 9-87–89, 9-94,

9-117, 9-138–9, 9-148, 9-154–5, 9-163
memory 1-1, 1-5, 1-8–9, 2-1, 2-6, 2-16, 3-7, 3-10, 9-1, 9-17, 9-20, 9-24, 9-26, 9-28, 9-34, 9-36, 9-38–39, 9-46, 9-50, 9-56, 9-58, 9-84, 9-86–88, 9-90, 9-92, 9-94, 9-96, 9-100, 9-118, 9-132, 9-142, 9-154, 9-161, C-1
VLSI 1-6

W

Wait state 1-5, 4-5, 7-6, 7-10–16, 8-4, C-7
Watch CPU Writes 7-4–5, 7-15, 8-3, C-3
Watch I/O Reads 7-4, 7-14, 8-3–4
Watch I/O Writes 7-4, 7-14, 8-3–4, C-3
Workstation 1-1

Z

Zero-extended 2-2

FAIRCHILD
A Schlumberger Company

U.S. and Canadian Sales Offices

ALABAMA

Huntsville Office
555 Sparkman Drive, Suite 1030
Huntsville, Alabama 35805
Tel: 205-837-8960

ARIZONA

Phoenix Office
9201 North 25th Avenue, Suite 215
Phoenix, Arizona 85021
Tel: 602-943-2100

CALIFORNIA

Auburn Office
320 Aeolia Drive
Auburn, California 95603
Tel: 916-823-6664

*Costa Mesa Office
3505 Cadillac Avenue, Suite O-014
Costa Mesa, California 92626
Tel: 714-241-5900

Encino Office
Crocker Bank Building
15760 Ventura Boulevard, Suite 1027
Encino, California 91436
Tel: 818-990-9800

*Cupertino Office
10400 Ridgeview Court
Cupertino, California 95014
Tel: 408-864-6200

San Diego Office
4355 Ruffin Road, Suite 100
San Diego, California 92123
Tel: 619-560-1332

COLORADO

Denver Office
10200 East Girard, Bldg B, Suite 222
Denver, Colorado 80231
Tel: 303-695-4927

CONNECTICUT

Woodbridge Office
131 Bradley Road
Woodbridge, Connecticut 06525
Tel: 203-397-5001

FLORIDA

Ft. Lauderdale Office
450 Fairway Drive, Suite 107
Deerfield Beach, Florida 33441
Tel: 305-421-3000

Altamont Springs Office
Crane's Roost Office Park
399 Whooping Loop
Altamonte Springs, Florida 32701
Tel: 305-834-7000

GEORGIA

Norcross Office
3080 Northwood Circle #130
Norcross, Georgia 30071
Tel: 404-441-2740

ILLINOIS

Itasca Office
500 Park Boulevard, Suite 575
Itasca, Illinois, 60143
Tel: 312-773-3133

INDIANA

Indianapolis Office
7202 North Shadeland, Room 205
Castle Point
Indianapolis, Indiana 46250
Tel: 317-849-5412

IOWA

Cedar Rapids Office
373 Collin Road N.E., Suite 200
Cedar Rapids, Iowa 52402
Tel: 319-395-0090

KANSAS

Overland Park Office
8600 West 110th Street, Suite 209
Overland Park, Kansas 66210
Tel: 913-451-8374

MARYLAND

Columbia Office
2000 Century Plaza, Suite 114
Columbia, Maryland 21044
Tel: 301-381-2500

MASSACHUSETTS

*Waltham Office
1432 Main Street
Waltham, Massachusetts 02154
Tel: 617-890-4000

MICHIGAN

Farmington Hills Office
21999 Farmington Road
Farmington Hills, Michigan 48024
Tel: 313-478-7400

MINNESOTA

*Minneapolis Office
360 West 80th Street, Suite 590
Bloomington, Minnesota 55431
Tel: 612-835-3322

NEW JERSEY

New Jersey Office
Vreeland Plaza
41 Vreeland Avenue
Totowa, New Jersey 07512
Tel: 201-256-9011

NEW MEXICO

Albuquerque Office
North Building
2900 Louisiana N.E., Suite D
Albuquerque, New Mexico 87110
Tel: 505-884-5601

NEW YORK

Endwell Office
421 East Main Street
Endicott, New York 13760
Tel: 607-757-0200

Fairport Office
815 Ayrault Road
Fairport, New York 14450
Tel: 716-223-7700

Hauppauge Office
300 Wheeler Road
Hauppauge, New York 11786
Tel: 516-348-0900

Poughkeepsie Office
19 Davis Avenue
Poughkeepsie, New York 12603
Tel: 914-473-5730

NORTH CAROLINA

Raleigh Office
5970-C Six Forks Road
Raleigh, North Carolina 27609
Tel: 919-848-2420

OHIO

Cleveland Office
6133 Rock Slide Road, Suite 407
Cleveland, Ohio 44131
Tel: 216-447-9700

Dayton Office
7250 Poe Avenue, Suite 260
Dayton, Ohio 45414
Tel: 513-890-5813

OREGON

Portland Office
6600 S.W. 92nd Avenue, Suite 27
Portland, Oregon 97223
Tel: 503-244-6020

PENNSYLVANIA

*Willow Grove Office
Willow Wood Office Center
3901 Commerce Avenue, Suite 110
Willow Grove, Pennsylvania 19090
Tel: 215-657-2711

TEXAS

Austin Office
8240 Mopac Expressway, Suite 270
Austin, Texas 78759
Tel: 512-346-3990

*FAIRTEC Design Center

FAIRCHILD
A Schlumberger Company

U.S. and Canadian Sales Offices

*Richardson Office
1702 North Collins Street, Suite 101
Richardson, Texas 75081
Tel: 214-234-3811

Houston Office
9896 Bussonnet-2, Suite 470
Houston, Texas 77036
Tel:713-771-3547

UTAH

Salt Lake City Office
5282 South 320 West, Suite D120
Salt Lake City, Utah 84107
Tel: 801-266-0773

WASHINGTON

Bellevue Office
11911 N.E. First Street, Suite 310
Bellevue, Washington 98005
Tel: 206-455-3190

CANADA

Toronto Regional Office
2375 Steeles Avenue West, Suite 203
Downsview, Ontario M3J 3AB, Canada
Tel: 416-665-5903

Montreal Office
3675 Sources Boulevard, Room 109
Dollard des Omeaux
Quebec, Canada H9B 1T6
Tel: 514-683-0883

Ottawa Office
148 Colonnade Road S., Unit 3
Nepean, Ontario K2E 7J5 Canada
Tel: 613-226-8270

*FAIRTEC Design Center

FAIRCHILD
A Schlumberger Company

International Sales Offices

AUSTRALIA

Fairchild Australia Pty. Ltd.
Suite 1, 1st Floor
366 Whitehorse Road
Nunawading 3131
Melbourne, Victoria
Australia

AUSTRIA

Fairchild Semiconductor GmbH
Assmayergasse 60
A-1120 Vienna, Austria
Tel: 43-222-858682
Telex: 115096

BENELUX

Fairchild Semiconductor B.V.
Ruysdaelbaan 35
NL-5613 DX-Eindhoven
The Netherlands
Tel: 31-40-446909
Telex: 51024

ENGLAND

***Fairchild Semiconductor Ltd.**
European Research, Design and
Applications Centre
Shire Hall, Shinfield Park
Reading
Berks. England
Tel: 44-734-752175
Telex: 849317

Fairchild Semiconductor Ltd.
230 High Street
Potters Bar,
Herts. England
Tel: 44-707-51111
Telex: 262835

FRANCE

Fairchild Semiconductor S.A.
12 Place des Etats-Unis
92120 Montrouge (Paris), France
(Mailing Address)
B.P. 655
92542 Montrouge Cedex, France
Tel: 33-1-47466161
Telex: 200614

HONG KONG

Fairchild Semiconductor Products Ltd
12th Floor, Austin Tower
22-26A Austin Avenue
Tsimshatsui
Kowloon, Hong Kong
Tel Nbr: (3) 7235256

ITALY

Fairchild Semiconductor S.p.A.
Viale Corsica 7
20133 Milan, Italy
Tel: 39-2-7491271
Telex: 330522

Fairchild Semiconductor S.p.A.
Via Francesco Saverio Nitti 11
00191 Rome, Italy
Tel: 39-6-328-7548 or 328717
Telex: 612046

JAPAN

Fairchild Japan Corporation
7th Floor Pola Shibuya Bldg.
15-21 Shibuya 1-Choma Shibuya-K
Tokyo 150, Japan
Tel: 81/3/4008351

Fairchild Japan Corporation
Yotsubashi, Cyhuo Building
4-26 Shinmachi 1-Chome
Nichi-Ku, Osaka, 550, Japan
Tel: 81/6/5416138

KOREA

Fairchild Electronics Ltd.
Korea Branch
10th Floor, Life Building
61 Yuido-dong, Youngdongpu-ku
Seoul 150, Korea

SINGAPORE

Fairchild Sales Pte. Ltd.
75/77 Bukit Timah Road #03-01/02
Boon Siew Building
Singapore 0922
Republic of Singapore

SWEDEN

Fairchild Semiconductor AB
Bergsundsstrand 39
S-11738 Stockholm, Sweden
Tel: 46-8-840170
Telex: 17759

SWITZERLAND

Fairchild Semiconductor GmbH
Baumackerstr. 46
CH-8050 Zurich, Switzerland
Tel: 41-1-3114230
Telex: 823285

TAIWAN

Fairchild Electronics (Taiwan) Ltd.
Hsietsu Building Room 502, 5th Floor
47 Chung Shan N. Road, Sec. 3
Taipei, Taiwan

WEST GERMANY

Fairchild Semiconductor GmbH
Flughafen Franchtz. Geb. 458
D-6000 Frankfurt/M. 75
West Germany
Tel: 049-69-6905613
Telex: 411829

Fairchild Semiconductor GmbH
Oeltzenstr. 14
D-3000 Hannover, West Germany
Tel: 49-511-17844
Telex: 922922

Fairchild Semiconductor GmbH
Postrasse 37
D-7250 Leonberg, West Germany
Tel: 049-7152-41026
Telex: 7245711

Fairchild Semiconductor GmbH
Zweigniederlassung Neufahrn
Hanns-Braun-Strasse 50
D-8056 Neufahrn (Munich)
West Germany
Tel: 49-8165-6180
Telex: 526770

*European Design Center

102650210

Notes:

INTERGRAPH
Advanced Processor Division

Embarcadero Place
Building D
2400 Geng Road
Palo Alto, CA 94303

This manual provides complete information on the CLIPPER 32-bit microprocessor, the microprocessor that incorporates proven supercomputer and mainframe architectural design techniques to bring mainframe performance to desktop computing. The CLIPPER microprocessor is implemented in three CMOS VLSI chips and includes a CPU/FPU and two combined memory management/cache units, one for data and one for instructions.

Extensively supplemented with hardware and software design examples and guidelines, this comprehensive manual will serve as a valuable resource for design engineers, computer architects and programmers in the development of high performance CLIPPER-based systems. It will also be helpful to engineers investigating the CLIPPER microprocessor family, and to students learning about state-of-the-art computer architectures.

Chapter 1 provides an overview of computer architectural concepts and how they influenced the design of the CLIPPER microprocessor. Chapters 2–6, 9, and 10 will be of particular interest to software engineers. They describe the programming model, memory management, caches, instruction pipeline, floating-point unit, software design, and the CLIPPER instruction set. Chapters 7 and 8 are intended for hardware designers and provide details of the CLIPPER bus as well as hardware design guidelines.

Fairchild reserves the right to make changes in circuitry or specifications at any time without notice. Fairchild cannot assume responsibility for use of any circuitry described other than circuitry embodied in a Fairchild product. Printed in U.S.A.

AD765021.01

PRENTICE-HALL, INC.
Englewood Cliffs, N.J. 07632

ISBN 0-13-138058-3